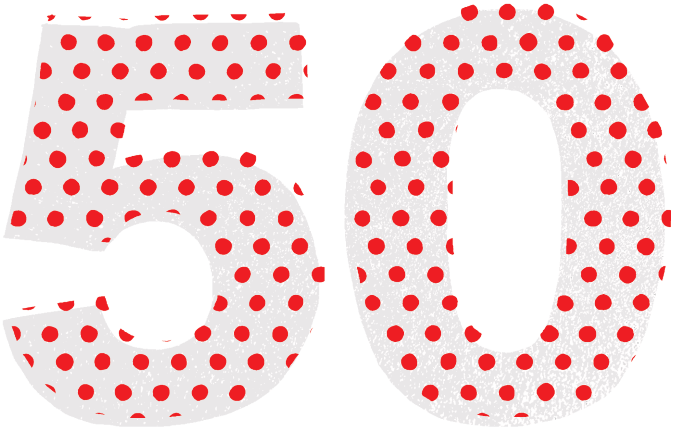
TYPESCRIPT IN 50 LESSONS

by

STEFAN BAUMGARTNER

TYPESCRIPT



LESSONS

by

STEFAN BAUMGARTNER

Published 2020 by Smashing Media AG, Freiburg, Germany.

ISBN: 978-3-945749-90-6

Cover and interior illustrations: Rob Draper
Copyediting: Owen Gregory
Cover and interior layout: Ari Stiles
Ebook production: Cosima Mielke
Typefaces: Elena by Nicole Dotin, Mija by Miguel Hernández and Andalé Mono by Steve Matteson

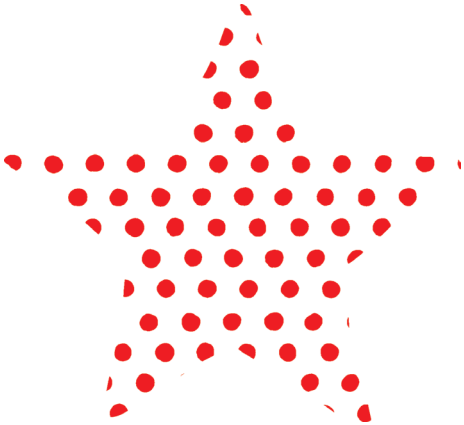
TypeScript in 50 Lessons was written by Stefan Baumgartner and reviewed by Shawn Wang.

This book is printed with material from FSC® certified forests, recycled material and other controlled sources.

Please send errors to: errata@smashingmagazine.com

To Doris, Clemens, and Aaron

Table of Contents



Acknowledgements

This book would not exist if it wasn't for Markus Seyfferth. Not only did he encourage me to pursue my book idea, but our regular meetings helped tremendously to shape the outline and contents of the final piece.

My ScriptConf buddies – Sebastian Gierlinger and Dominik Angerer – are partners in crime, always open to validate ideas, and invaluable friends. Their feedback and motivation helped a great deal to finalize this book.

I want to thank Rob Draper, who not only created the marvellous artwork you see in this book, but also helped a lot to find a theme and put a very technical topic in a very human perspective.

The wonderful crew at Smashing. It has been such a joy to work with you on this piece. Rachel Andrew is the best editor one can wish for. Owen Gregory makes me sound wonderfully English. Ari Stiles with her captivating enthusiasm creates art out of words. Cosima Mielke, who applied every ebook trick available to make the final result extraordinary. And Vitaly Friedman is constantly helping me to make my ideas a reality.

To say Shawn Wang did the technical review would not do his influence justice. He put a lot of TypeScript culture into the book that made it complete.

My colleagues at Dynatrace, especially Fabian Friedl, Thomas Heller and Ernst Ambichl. Working with them had direct influence on every page written.

The TypeScript and TSConf:EU community. Daniel Rosenwasser, Anders Hejlsberg, Marius Schulz, Natalie Marleny, Bert Belder, Fred Schott, Lili Kastilio, Nathalia Rus, Vanessa Böhner, Marvin Hagemeister, Gary Bernhardt, Mirjam Bäuerlein, Georg Kothmeier, Alex Rosemann, and Peter Kröner all had direct or indirect influence on the book.

When you think of the TypeScript community, you inevitably think of Orta Therox, whose tireless work in and out of hours continuously brings the community together. I'm incredibly honored that he agreed to open this book.

Last, but not least, I want to thank my wonderful family – Doris, Clemens, and Aaron – for their support, their understanding, and their shared enthusiasm. And I'm sorry for all the times my mind wandered off into type systems, where it should've been with the LEGO blocks.

Foreword

There are so many ways to learn a programming language, and even more ways to go from "I can read this" to "I truly understand what is going on here."

The official TypeScript documentation has to walk a fine line between being authoritative and trying to cater to a large amount of learners with diverse backgrounds. As more people adopt TypeScript, we need more ways to help them understand the tools the language provides.

When I interviewed for the TypeScript team, I pitched a vision of language documentation that takes into account the rich community of people writing articles, who help the TypeScript team's writing stay focused.

Stefan Baumgartner is one of those incredible community members.

Stefan runs TypeScript meetups and conferences, and his blog is a constant source of delightful, insightful articles on the language.

His articles have a personal tone, which reminds me of DMing a colleague a question and they respond with: "Give me a second, I'll come by and explain this properly." This

book takes that down-to-earth approach and hits all the foundational points in the language. It's a great fit for Smashing.

TypeScript is an evolving language, and the new features can sound so obtuse unless you have a firm understanding of the foundations and the design constraints on the language. That's why I was so happy to see the interludes in this book which help fill in some of the gaps of *why* instead of just showing you the *how* of TypeScript.

So, I'll raise an emoji 🏆 to a great resource for people growing their knowledge and taking those first few steps into that uncomfortable but gratifying space of "Today, I learned."

—Orta Therox

Introduction

Do you have some time to talk about TypeScript? If you have been following discussion in the tech community during the last couple of years, there has been no way of avoiding countless people gushing over their newest toy: TypeScript, a programming language atop of JavaScript, which supposedly makes everything a lot better.

The flood of information on TypeScript, and the amount of opinions on TypeScript, can be overwhelming. But there is no denying that its significance grows stronger every day. In the 2020 StackOverflow developer survey,[1] 67% stated that TypeScript was their most loved language, coming in at second place. While with npm, Inc., Laurie Voss found in the 2019 npm survey[2] that around 63% of npm users used some sort of TypeScript, with 52% primarily writing TypeScript. Those are big numbers!

The same person, a different survey, a different community: in the "State of the Jamstack Survey 2020,"[3] Laurie found that around 10% use TypeScript primarily. About this survey, Laurie said:

1 https://smashed.by/2020survey
2 https://smashed.by/npmsurvey
3 https://smashed.by/jamstacksurvey

> *I keep running into this phenomenon that people who write TypeScript think of themselves as TypeScript devs, not JavaScript developers. Which is strange because TypeScript itself describes itself as being… just JavaScript!*

Laurie has a point here. There seems to be an artificial split between people who consider themselves JavaScript developers, and people who think they don't write JavaScript anymore. I myself kept ditching TypeScript[4] for years, seeing it only as a way to make something supposed to be easy – JavaScript – a lot more difficult. To me, TypeScript was JavaScript for Java developers.

Oh, how wrong I was! TypeScript can be much more than that. A subtle tool, a simple layer atop the programming language that drives the web. If you ever find yourself:

- writing JavaScript with libraries and frameworks you barely know
- writing JavaScript with others
- writing JavaScript that deals with back-end data
- writing JavaScript that your future self has to continue working on

4 https://smashed.by/whytypescript

then TypeScript will do right by you. This book will give you a gentle, human introduction to one of most beloved programming languages, and you will end up a master of type systems.

Who This Book Is For

This book is for developers who know enough JavaScript to be dangerous. They spend an increasing amount of time programming and want to be more productive in doing so. With TypeScript, they hope to get more information out of their JavaScript code – for themselves and their colleagues.

This book is also for developers who dipped their toes into TypeScript and now want to get their feet wet. They want to learn about type systems and how they can be used to define complex JavaScript scenarios. This knowledge will ultimately become language-independent, preparing them for different programming languages that have elaborate type systems.

Scope of This Book

Programming books have a tendency to become outdated very quickly. The moment you hold the printed version in

your hands, the world has moved on and parts are out of date, or important lessons are left out. When I set out to write this book, my most important goal was that it had to be timeless. TypeScript gets at least two major releases a year, so there are new features and changes on certain aspects of the programming language.

That's why we focus on the long-lasting aspects of the type system. The main way to program will be JavaScript; TypeScript will work as an additional type layer describing the behavior of our code. This is also aligned with the way the TypeScript team design their upcoming work. After reading this book, you will immediately understand what new features are about.

TypeScript in 50 Lessons

This book is TypeScript for humans, so I want to give you the most human introduction to the programming language. This is why we split up the book into seven chapters with seven lessons each, with a final lesson at the end.

The lessons are practical, based on real-world problems that I and many friends and peers have encountered over the last few years. Each lesson takes no longer than ten min-

utes to read and digest, making it the perfect companion for commuting, or a little light reading before bed. Smaller, casual interludes between chapters provide context or point to TypeScript features outside our main focus.

On the book's website,[5] you will find editable examples for each lesson for you to play around with.

Chapters build on one another, with each chapter focusing on a specific part of the language around a concrete example.

Here's a chapter rundown.

TypeScript for Smashing People
We go on a hunt for red squiggly lines. If a word processor can highlight our spelling and grammar mistakes, why shouldn't a programming editor do the same? In this opening chapter, we will see that – given the right tools – we might already be using TypeScript without realizing. With TypeScript being a gradual type system, we can gently encourage the programming language to give us more insights into our code. We will also write our first types.

Working with Types
We learn about some major features of TypeScript, like type annotations, type inference, and control flow. We will define

5 https://typescript-book.com

primitive and complex types, and learn about the difference between types and interfaces. For every variable or constant we can create, we find a way to provide a type.

Functions
Functions are an essential feature in JavaScript, and we can see that once we want to type function signatures.
We learn about function heads and bodies, structural typing for functions, and how we can define different behavior for the same function.

Union and Intersection Types
TypeScript's type system can be seen as an endless space of values, and types are nothing but discrete sets of values inside this space. This allows for algebraic operations like union and intersections, making it a lot easier for us to define concrete types for values. We learn about type widening and narrowing, top and bottom types, and how we can influence control flow.

Generics
Generics are a way to prepare types for the unknown. Whenever we know a certain behavior of a type but can't exactly say which type it affects, a generic helps us to model this behavior. We learn about generic constraints, binding generics, mapped types, and type modifiers.

Conditional Types
Conditional types are arguably the most unique feature to TypeScript's type system. They allow us to introduce a level of meta-programming unseen in programming languages, where we can create if/else clauses to determine a type based on the input type. This allows for a powerful set of tools we can use to define model and behavior once, and make sure we don't end up in type maintenance hell.

Thinking in Types
The final chapter deals with situations you might encounter in your everyday programming life. We use these situations to get into a thinking-in-types mindset, where we take care about a robust and well-defined set of types before starting implementation. This helps us validate that what we code is what we expect.

Chapter One

TYPESCRIPT FOR SMASHING PEOPLE

TypeScript for Smashing People

In this chapter, we want to debunk myths. TypeScript can be so many things, and many people have different views on this programming language that has become so popular in recent years. What is TypeScript actually about? Let's see what TypeScript has in store for us.

Lesson 1: Red Squiggly Lines

Take a look at this piece of code.

```
const storage = {
  max: undefined,
  items: []
}

Object.defineProperty(storage, 'max', { readonly:
true, val: 5000 })

let currentStorage = 'undefined'

function storageUsed() {
  if(currentStorage) {
    return currentStorage
  }
  currentStorage = 0
  for(const i = 0; i < storage.length(); i++) {
    currentStorage += storage.items[i].weigth
  }
```

```
  return currentStorage
}

function add(item) {
  if(storage.max - item.weight >= storageUsed) {
    storage.items.add(item)
    currentStorage += iten.weight
  }
}
```

Just about 25 lines of JavaScript and a lot is going on in there! If we investigate this snippet, we might deduce that it's about some storage, maybe of a ship, that can hold about 5,000 tons of items. Two functions help us by adding items and checking if there is still room left.

Looks pretty easy, doesn't it? Well, there's a catch: a sneaky little error hidden somewhere within those 25 lines of code. Can you spot it?

All right, that was a little lie. There's not just one error, there are lots. There are so many mistakes and errors that not a single line is without flaws. This piece is utterly broken. And it gets worse. We don't get any feedback from the browser (or Node.js for that matter) telling us that something is off. This snippet neither breaks nor fills the

browser's console with error messages. It stays quiet when we call the add method:

```
add({ weight: 3000 })
```

No visible errors. It's just not working.

Welcome to JavaScript! Things like this are known to drive developers mad. People doubting their career choices, spending hours finding out where their finger slipped to the neighboring key, making a usually solid piece of work erroneous. Such errors also give JavaScript a bad reputation, which is sad for a programming language that enables so many of us to create wonderful things.

The longer you stare at the 25 lines of JavaScript the more problems you will find. Things that are outright broken, but also things that seem only slightly off and invite you to test around. The problem is, just by looking at the text, it's hard to keep track of everything that might be out of the ordinary.

Wouldn't it be nice if we could get some visual feedback helping us to identify problems immediately, instead of having to spend hours looking for them? Think of a wrod – excuse me – word processor showing you typos with red squiggly lines.

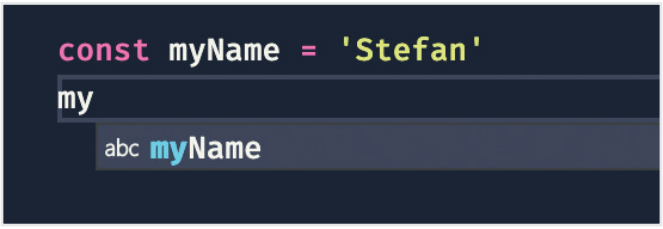
Enter TypeScript

This is the purpose of TypeScript: pointing out potential errors before your code hits the production environment. First and foremost, TypeScript provides code analysis for JavaScript.

Take a modern code editor. Maybe you use Visual Studio Code (VS Code).[6] When you open a JavaScript file, VS Code might not look too different from other editors. But underneath, TypeScript is already active. Running, inspecting, analyzing. Need proof? Create a new JavaScript file and enter the following line:

```
var myName = 'Stefan'
```

You are invited to use your name, of course. Now enter "my", hold the **Ctrl** key on your keyboard and press **Space**. You should get a view like this:



Editors pick up names and give you autocompletion.

6 https://code.visualstudio.com/

VS Code tries to autocomplete your writing with a variable name it already found: `myName`. To get this variable name, TypeScript ran through your file and tried to collect every name it could find, so you can reuse them when writing.

> Even though VS Code gives us TypeScript out of the box, you can have the same features in your editor of choice. Look for plug-ins called "TypeScript language server." The TypeScript language server implements a protocol that can fetch code from the editor and returns analysis feedback. This gives you red squiggly lines in almost every editor available.

So TypeScript is active, but it's also a little shy. To unleash TypeScript's full power, we have to explicitly invite TypeScript to tell us the results of the analysis. By adding `//@ts-check` as the very first line of your file, TypeScript will start to add red squiggly lines to code pieces that just don't make sense:

```
//@ts-check
const storage = {
  max: undefined,
```

```
  items: []
}

Object.defineProperty(storage, 'max', { readonly: true,
val: 5000 })

let currentStorage = 'undefined'

function storageUsed() {
  if(currentStorage) {
    return currentStorage
  }
  currentStorage = 0
  for(const i = 0; i < storage.length(); i++) {
    currentStorage += storage.items[i].weigth
  }
  return currentStorage
}

function add(item) {
  if(storage.max - item.weight >= storageUsed) {
    storage.items.add(item)
    currentStorage += iten.weight
  }
}
```

Oh, wow! Seems like there are a couple of problems! Some of them seem obvious, others not. Let's investigate them one by one.

Lesson 2: Hunting Bugs

With the addition of `//@ts-check`, TypeScript became active in our JavaScript file and showed the first list of problems. Usually, code editors not only give visual feedback in the form of red squiggly lines but also let you know what the problem is. When you hover over one of the problematic pieces, VS Code gives you an explanation in a pop-up. But you might notice the problems yourself once they're highlighted. The more you work with TypeScript, the more intuitive it becomes.

readonly Is Not Assignable

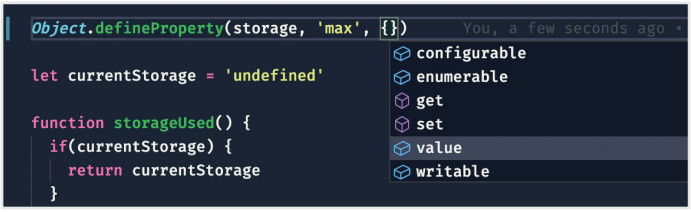
At line 7 we want to define a new property called `max`, set the value to 5,000 and make sure we can't overwrite the value. TypeScript complains that the property `readonly` is not assignable to a `PropertyDescriptor`. Fancy words!

What they mean is that we've mixed up words. Property descriptors don't know of a property called `readonly`; it's called `writable`. Instead of a `readonly` value of `true`, we need a `writable` value of `false`. When we correct our mistake, TypeScript will also tell us that `val` does not exist – it's `value`. The corrected line looks like:

```
Object.defineProperty(storage, 'max', { writable:
false, value: 5000 })
```

This is the first feature of TypeScript: making sure you use the correct names for the things you want to use. No more typos, no more wrong spellings or mixed up terms. We call this a **type check**: making sure you deliver what's expected.

What's the difference between `readonly: true` and `writable: false` anyway? One is understood by JavaScript, the other isn't. But how are we supposed to know which properties to set? Since TypeScript knows which properties to expect and gives you an error when you go wrong, we can use the same information to get editor assistance. Press **Ctrl + Space** right inside the object and you again get autocomplete, but within the context of your current line:



With TypeScript active, we immediately get details of what information we expect at this particular point in our code.

Not only that, but we can also make sure we don't add something other than `true` or `false` when adding a value to `writable`.

Type Inference

On to the next red squiggly line. When we hover over the red lines under `currentStorage`, we see that "Type 0 is not assignable to type 'string.'" In JavaScript, you can assign different values to variables, including completely unrelated values and values of different types. JavaScript doesn't care if your `let foo` is "Garfield" (a string of characters), 1337 (a number), or `{ heavy: true }`.

This is also the moment when most errors arise. Once you assign a value to a variable, you most likely want it to keep a specific type. A couple of lines earlier, we created our variable `currentStorage` and set it to `'undefined'`.

However, we made a little error there. We assigned the string value "undefined" to it, not the programmatic `undefined`. This `undefined` tells JavaScript that there is neither a type nor a value yet.

```
// this should be currentStorage = undefined
let currentStorage = 'undefined'
```

```
function storageUsed() {
  if(currentStorage) {
    return currentStorage
  }
  currentStorage = 0
  for(const i = 0; i < storage.length(); i++) {
    currentStorage += storage.items[i].weigth
  }
  return currentStorage
}
```

This causes a cascade of errors! For example, `if(current Storage)` evaluates to true before we could even set the actual current storage amount. If we set `currentStorage = undefined`, this condition evaluates to false. This in turn leads to the code where we sum up the current storage amount for the very first time (don't worry, this part has tons of errors as well).

TypeScript warns us that we're mixing types. When initializing `currentStorage`, we assign a string value. Later on, we want it to be a number. This is usually behavior we don't want, so TypeScript throws errors at us.

TypeScript uses a concept called **type inference**. The moment we assign a value to a variable, TypeScript tries to infer the type from the assignment. `currentStorage = 0`, for example, tells TypeScript that `currentStorage` is expect-

ed to be a number. From that point on, we can only reassign numbers or do number-based things (mathematical operations, for instance).

The moment we assign `undefined`, `currentStorage` can be anything until it gets a distinct type. To solve our problem, we change `'undefined'` to `undefined`:

```
let currentStorage = undefined

function storageUsed() {
  // Suddenly this evaluates to false with the
first call
  if(currentStorage) {
    return currentStorage
  }

  // From now on, currentStorage is a number
  currentStorage = 0
  ...
  // and storageUsed() returns a number
  return currentStorage
}
```

Type inference also works within methods. In our example above, we return `currentStorage` at the end of `storageUsed()`. Since we know that `currentStorage` becomes a number, `storageUsed()` also has to return a number. In `const x = storageUsed()`, `x` will be a number.

Semantic Checks

On line 16 we have a for loop where we go through all our storage items and make a sum.

```
for(const i = 0; i < storage.length(); i++) {
  currentStorage += storage.items[i].weigth
}
```

Unfortunately, this loop will crash. The reason: we declared the initialization variable `i` as a constant. Constants can't be reassigned; that's why this code won't work. A change to `let` solves this problem:

```
for(let i = 0; i < storage.length(); i++) { // OK!
currentStorage += storage.items[i].weigth
}
```

This is one of many **semantic checks**. TypeScript not only tells us what's wrong. With the correct editor integration, it can also suggest quick fixes that solve your problem.

Again, the main purpose of TypeScript is to give you the best tooling possible. It wants to understand your code better than you do.

TypeScript knows of common errors and suggests automatic fixes.

The Last Bits

By now you should be familiar with the benefits TypeScript gives us: it analyzes code, tells us what's wrong, and returns suggestions on how we can prevent or fix potential errors.

It does so by comparing the shape of objects and variables from what it can infer to what's supposed to be. The last few bits fall into the same categories.

On line 23 TypeScript tells us that "Operator '>=' cannot be applied to types 'number' and '() => number'". Not only is the condition really hard to read, but we also compare numbers to a function! We forgot to call `storageUsed`, so let's fix that:

```
-if(storage.max - item.weight >= storageUsed)
+if(storage.max - item.weight >= storageUsed())
```

The next line tells us that `add` is not a valid method on arrays; it should be `push`:

```
-storage.items.add(item)
+storage.items.push(item)
```

Last, but not least, we have a nasty typo. Where does this `iten` come from, when it should be an `item`?

```
-currentStorage += iten.weight
+currentStorage += item.weight
```

With the simple addition of a comment line at the beginning of our JavaScript file, we were able to figure out a ton of potential problems and pitfalls that would have made our program crash.

TypeScript does this by comparing the eponymous types. But what are types? And can we use types to figure out even more problems? We can, we can!

Lesson 3: Types

We already found a couple of errors in our last lesson, but we are not done yet. There are many more problems to be found! To get to the bottom of the barrel, we have to be a little bit more deliberate about our types. Wait a second. What even are types?

In his book, *Programming with Types*,[7] Vlad Riscutia defines types as follows:

> *A **type** is a classification of data that defines the operations that can be done on that data, the meaning of the data, and the set of allowed values. Typing is checked by the compiler and/or run time to ensure the integrity of the data, enforce access restrictions, and interpret the data as meant by the developer.*

With types, we know that `"Hello World"` is a string of characters, `1234` is of type number, and `true` is Boolean. Without types like `boolean`, `string`, or `number`, those values would just be zeros and ones in some computer's memory, and we wouldn't know what to do with them.

7 https://smashed.by/programmingwithtypes

A type not only defines how a value should be interpreted, it also suddenly affords us operations! We can multiply 1,234 by 10, or make "Hello World" all uppercase with `.toUpperCase()`.

Types are fundamental to programming. JavaScript has types! And don't let anybody else tell you otherwise! In JavaScript, you have primitive types like `number`, `string`, and `boolean`. Yes, there are other primitive types, and we will come to them in time.

There are also composite data types like:

- **objects**: properties and values of different primitive and composite types
- **arrays**: lists of values which can take any type
- **functions**: methods you call with parameters of certain types, and which return values of certain types

And there are symbols, but they are a whole different story.

Weakly Typed

JavaScript, however, is weakly typed. This means that even though you create a variable or property and assign a value of a certain type, you can switch types on the run:

```
let val = 1234; // OK!
val = 'Onetwothreefour'; // Reassignment. Still OK!
```

You can also combine values of different types without a hitch:

```
let val = { a : 3 } + 5
// [object Object]5! What does that even mean?
```

The results don't necessarily have to make sense. And even though operations like that are possible, they're widely considered errors – errors that aren't caught.

Strongly Typed

TypeScript is strongly typed. This means that once you assign a value of a certain type, TypeScript wants you to stick with it.

```
let val = 1234; // OK! val is a number
// Wait, what? Now it's a string? This can't be!
val = 'Onetwothreefour';
```

This also means that you can't use operators on values of different types, as they usually don't make sense:

```
// You want to add a number to an object? Why??
let val = { a : 3 } + 5
```

And with that, you get all the red squiggly lines. If you are used to reusing variables or adding numbers and strings on the fly, you might feel a little bit restricted when using TypeScript.

You trade this flexibility for code that is a lot more sound, correct, and void of potential pitfalls.

Shapes

Primitive types are rather simple to define. They can have a very defined range of values. The type `boolean`, for example, is the simplest: it can be `true`, `false`, `undefined`, or `null`. You can count the number of possible values with your fingers. Numbers and strings allow for far more values, but only what memory and the current JavaScript runtime allow (check `Number.MIN_VALUE` and `Number.MAX_VALUE` to get a feel of the range number can take).

It gets more complex with composite types. Here, we deal not only with ranges of values, but also with so-called shapes. Shapes tell us more about the structural features

of a type: types and names of properties of an object, types and names of parameters of a function, types and indexes of elements in an array.

Take the following `person` object, for example:

```
const person = {
  firstName: 'Stefan',
  lastName: 'Baumgartner',
  age: 38
}
```

`person` is of type object, but follows the shape: `firstName` is a string; `lastName` is a string; and age is a number. We can define this shape as a custom type:

```
type Person = {
  firstName: string,
  lastName: string,
  age: number
}
```

With TypeScript being a *structural type system*, shapes are incredibly important. As long as variables match a certain shape (in the way that `{ writable: false }` matches the shape of `PropertyDescriptor` in our previous example),

TypeScript will be OK with your code. Should you provide objects with different shapes, you'll get even more errors. Let's check on that in the next lesson.

Lesson 4: Adding Types with JSDoc

Now that we know what types are, we can start to be a bit more intentional with the data objects and functions in our little script. Adding types to an existing JavaScript file can happen in many ways. One of the easiest is to use a little tool called JSDoc.

JSDoc[8] is a way to annotate our code using comments. We describe function signatures, object properties, and much more by using certain conventions:

```
/**
 * Adding two numbers. This annotation tells
TypeScript
 * which types to expect. Two parameters (params) of
 * type number and a return type of number
 *
 * @param {number} numberOne
 * @param {number} numberTwo
 * @returns {number}
 */
```

8 https://jsdoc.app/

```
function addNumbers(numberOne, numberTwo) {
  return numberOne + numberTwo
}
```

The JSDoc app usually runs over our annotated source code and creates HTML-based documentation. TypeScript uses the same annotations to get more information on our intended types.

```
// TypeScript throws an error here, because the JSDoc
// comments expect two numbers, not a number and
// a string
addNumbers(3, '2')

// TypeScript throws an error here, because addNumbers
// returns a number, and toUpperCase() is not available
// in number
addNumbers(3, 2).toUpperCase()
```

This is a great way to be very intentional about what types to expect. And with that intent comes safety when using functions and when implementing them.

```
/**
 * @param {number} numberOne
 * @param {number} numberTwo
```

```
 * @returns {number}
 */
function addNumbers(numberOne, numberTwo) {
  return numberOne.toUpperCase() + ''
  // Wait, what? We are treating numberOne like a
  // string, even though it's a number, and we return
  // it as a string even though we expect a number in
  // return, there's something wrong here!
}
```

As an added benefit, we get documentation with our types. Or do we get types with our documentation?

Custom Types

JSDoc works well with primitive types like `number`, `string`, and `boolean`. But we are also able to define composite types like objects and arrays with JSDoc.

Objects are a bit tricky. In JavaScript we can declare objects on the run with two little curly braces: `const x = {}`. This empty object has no extra properties at the moment, but this can change drastically as our program evolves. With `x.hey = 'Ho'`, we suddenly add a new property to `x` without any trouble.

Those freewheeling objects in JavaScript are flexible but also very unpredictable. That's why TypeScript doesn't

throw errors when an object property that hasn't yet been defined shows up in our regular JavaScript code. It could have been defined somewhere!

With a JSDoc type annotation, we can declare which object properties we expect to exist, and make sure that TypeScript knows what to expect. Suddenly, the object has a defined type – a contract – and we make sure that we always refer to this contract.

Let's add a JSDoc annotation to our storage object. Take a look at its current shape:

```
const storage = {
  max: undefined,
  items: []
}
```

There's a value called `max` that can be anything. undefined is a valid value of any type! And we have an array of items where we also don't yet know how our items should look.

Let's focus on items first. Items can have lots of properties, but the one we need according to our little script is weight, and weight should be of type number.

```
/**
 * @typedef {Object} StorageItem
```

```
 * @property {number} weight
 */
```

So within a comment, we define a new type called `StorageItem`, which is an object. It has one property called `weight`, which is a number. We also create a type for our storage object in the same fashion.

```
/**
 * @typedef {Object} ShipStorage
 * @property {number} max
 * @property {StorageItem[]} items
 */
```

Here, typing is very important as again we are very deliberate about what we expect. `max` is `undefined` for now, but it should be a number once we assign real values. And our array is full of `StorageItem`s, a custom type we defined just a couple of lines above. Now, with our types defined, let's apply them to the `storage` object:

```
/** @type ShipStorage */
const storage = {
  max: undefined,
  items: []
}
```

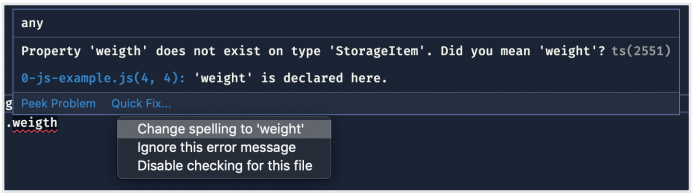
and find out where our code explodes!

Boom!

We get some troubles in our `storageUsed()` function:

```
function storageUsed() {
  ...
  for(let i = 0; i < storage.length(); i++) {
    currentStorage += storage.items[i].weigth
  }
  ...
}
```

First, the property `.length()` does not exist on type `Ship Storage`. It's a property on `items` that we get because `items` is an array. But it's not a function, it's a value. Also, `weigth` is a typo. TypeScript suggests a correction for us.



A TypeScript error not only shows us what's wrong, but also suggests what we meant to do! Finding and fixing typos is something TypeScript is really good at.

Intention

Just by adding a simple type annotation in a JSDoc comment, TypeScript knows so much more about the semantics of our program. And our intentions for variables, constants, and functions become a lot more visible. Types become a tool we can use throughout our code:

```
/**
 * @param {StorageItem} item
 */
function add(item) {
  if(storage.max - item.weight >= storageUsed()) {
    storage.items.push(item)
    currentStorage += item.weight
  }
}
```

Types signal intention, define a contract, and make sure we use our program code in the way it was supposed to be. And this comes at almost no cost. A little documentation layer on top of our regular JavaScript, and we get much more tooling, information, and knowledge.

JSDoc is very powerful and can carry you a long way. I've written an exhaustive reference of possible type annotations by comments[9] on my website.

9 https://smashed.by/jsdocsuperpowers

Lesson 5: Type Declaration Files

While JSDoc can get you very far, it can be a little unwieldy at times, especially when you want to define complex, nested object shapes, or want to reuse types over different files. You end up with a lot of comments and a lot of subtypes. Most likely, this will clutter your codebase more than it will help.

To make it easier for us to define custom types, describe global function interfaces, or share types between different parts of our JavaScript application, we can use type declaration files.

TypeScript, the Programming Language

This is also our first foray into a programming language that looks a lot like JavaScript but is in fact TypeScript. Up until now, we have used the tooling part of TypeScript: a type-checker, a language server that gives feedback to code editors. We haven't written something that couldn't be found in JavaScript's language specification.

What we are going to see now is new. Similar to JavaScript but different enough. And it can't be executed by a browser or Node.js runtime. TypeScript, the programming language. TypeScript calls itself a superset of JavaScript. That means TypeScript is a programming language that includes all of

JavaScript, and also more language constructs for additional features. One of them is defining custom object types.

Custom Type Declarations

We had a glimpse of custom object types in lesson 4. Let's see how we can define our `ShipStorage` type with this new programming language:

```
type StorageItem = {
  weight: number
}

type ShipStorage = {
  max: number,
  items: StorageItem[]
}
```

As you can see, this new declaration holds the same information as our JSDoc comments in lesson 4. But the way we define types looks entirely different – a little bit like how we define an object in JavaScript. If you compare `storage` from our JavaScript program, you can see the similarities:

```
type ShipStorage = {
  max: number,
```

```
  items: StorageItem[]
}

const storage = {
  max: 6000,
  items: []
}
```

With the type declaration, we can be very explicit about the shape of the objects we want to create. And if you compare the type and the actual object side by side, you see why we call it shape. As much as a baseball has the shape of a sphere, the `storage` object has the shape of a `ShipStorage` type.

We also say the type has a certain *structure*.

.d.ts Files

To make type declarations like this work, we have to put them into a TypeScript file. TypeScript supports type declaration files that end with *.d.ts*. Here, you can add all your custom types, but no extra program code.

We take the `ShipStorage` and `StorageItem` types from above and put it into a *types.d.ts* file that's somewhere next to your main JavaScript file.

Now we have our types in one location. They are, however, not yet available to our other files. To make our types available, we have to export them:

```
export type StorageItem = {
  weight: number
}

export type ShipStorage = {
  max: number,
  items: StorageItem[]
}
```

We get rid of all our comment-based type declarations in our main JavaScript files. Instead, we point to the exported types.

Right after your `@ts-check` comment, add the following two lines:

```
/** @typedef { import('./types.d').ShipStorage }
ShipStorage */
/** @typedef { import('./types.d').StorageItem }
StorageItem */
```

The syntax is very similar to our previous type definition. But instead of telling TypeScript – or JSDoc – that we're defining an object, we point directly to the object we already defined somewhere else.

We have just written our first custom types in TypeScript!

Please note that our JavaScript code hasn't changed at all; we still write regular JavaScript for the actual core of our program. TypeScript's type definitions simply exist on the side, making your life easier. This is one of the fundamental principles of TypeScript: making it as easy as possible to add types to your programming code.

So far you haven't needed any tools, just an editor that knows how to handle TypeScript. This leads to a gradual and unobtrusive workflow that allows you to add TypeScript without committing too much on tools and build processes.

1. With `//@ts-check` we activate TypeScript in the file we are currently editing.
2. We use JSDoc comment type annotations for all our constants, objects, and functions. Functions, especially, benefit a lot from the extra information.
3. We create custom type definitions in type declaration files, and load them as needed in our JSDoc type annotations.

This way of developing TypeScript is not uncommon. It's a great way to gradually migrate to TypeScript, yet also enough to get most of the benefits of TypeScript without alienating contributors by introducing a new programming

language. One popular framework that uses this approach is Preact.[10] Preact's entire codebase builds on added types!

Lesson 6: Ambient Declaration Files

Let's adjust our `addItem` function with a tiny little detail: during development, we want to log the current amount in our `storage` to the console. Debugging becomes easier, as we get constant feedback on how `storage` changes.

To make sure we only log during development, we create a global `isDevelopment` flag, which can take a value of the type `boolean`. If this value is set to `true`, we log; otherwise, we don't.

During development we include our JavaScript in a specially prepared HTML file that sets this particular flag to `true`:

```
<script>
const isDevelopment = true
</script>
```

In our production HTML we omit this piece, or set `isDevelopment` to `false`:

10 https://preactjs.com/

```
<script>
const isDevelopment = false
</script>
```

We now have a globally defined constant we can easily access from anywhere in our script. Depending on the environment in which we include our JavaScript files, we get a different output. Our adjusted `addItem` function looks something like this:

```
/**
 * @param {StorageItem} item
 */
function add(item) {
  if(storage.max - item.weight >= storageUsed()) {
    storage.items.push(item)
    currentStorage += item.weight
  }
  if(isDevelopment) {
    const itemCount = storage.items.length
    console.log(`${itemCount} items`)
    console.log(`${currentStorage} kg total`)
  }
}
```

This is a very common pattern. And if you write regular JavaScript, you probably have relied on *some* global constant or variable.

But TypeScript reports errors! Red squigglies under `isDevelopment`, because `isDevelopment` isn't defined anywhere. TypeScript can't find this particular name.

Custom Ambient Declarations

And TypeScript's right! The mere existence of `isDevelopment` relies on our goodwill.

TypeScript couldn't figure out what to expect. Is it a Boolean, a string, a composite object, function, or only undefined? We know at the moment that it's a Boolean. But will our co-workers, three months ahead?

To make globals known and defined, we can use ambient type declarations. These types are encompassing, existing and present on all sides.

We need another *.d.ts* file to put somewhere near our types, where we can define the function heads, global objects, and variables that we need throughout our program.

Let's create an *ambient.d.ts* file next to our main JavaScript file. We'll add one line in there:

```
declare const isDevelopment: boolean
```

We are again in TypeScript language territory. And again, you can see some similarities to JavaScript. `const isDevelopment` is the part that's taken directly from JavaScript. The `declare` keyword in front of it tells TypeScript that we want to make known that this constant exists. As it's just a declaration, we don't need to add concrete values.

The part after `const isDevelopment` is a type annotation, and a little spoiler to chapter 2. With `: boolean` we tell TypeScript that `isDevelopment` is – well – Boolean! Save this line, and `isDevelopment` becomes available in all your JavaScript files.

Installing Ambient Library Declarations

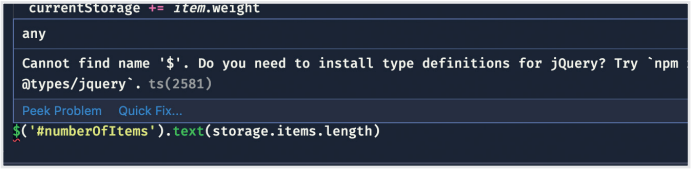
Ambient declarations are not only useful for global flags but also if you have functions and objects that you expect to exist. One example would be jQuery! Yes, the one JavaScript library that has taken web development by storm. And which is still widely used by millions but has somehow fallen from grace in the last couple of years.

But hey, many people owe their careers to jQuery. So let's show a little gratitude and use it for demo purposes in our simple ship's storage program.

Say we want to update the number of items on a web display every time we add a new item to our ship's storage. A little jQuery snippet in `addItem` would do the trick:

```
/**
 * @param {StorageItem} item
 */
function add(item) {
  if(storage.max - item.weight >= storageUsed()) {
    storage.items.push(item)
    currentStorage += item.weight
  }
  $('#numberOfItems').text(storage.items.length)
}
```

We again get the same error: TypeScript can't find `$`, the jQuery shortcut. But TypeScript is also very clever. Adding jQuery is very common, so TypeScript already hints at installing jQuery's types:



Coming across $ prompts TypeScript to suggest installing jQuery's types.

TypeScript has a big community of people who contribute custom types to almost any library that hasn't been written

in TypeScript but would benefit from type information. With jQuery still being widely used, it was one of the first libraries to get this special treatment.

All type definitions are available on npm, the JavaScript package registry. Node.js and npm are vital parts of a web developer's tool set.

We won't go into the details of how to work with Node.js and npm, but Jamie Corkhill's "Get Started With Node: An Introduction To APIs, HTTP And ES6+ JavaScript" over at Smashing Magazine[11] has you covered.

So, open your favorite terminal (hint: VS Code has one integrated) and install jQuery's types via npm in your current project's folder:

```
npm i @types/jquery
```

And just like that, you get full auto-completion and type-checking for jQuery in your JavaScript files.

jQuery's type declarations are ambient, and TypeScript can work with them. You have to make sure that jQuery exists when you roll out your application.

11 https://smashed.by/nodeintroduction

Lesson 7: Tooling

Adding TypeScript to your development workflow was designed to be as unobtrusive as possible. If you have a TypeScript-focused editor like VS Code, adding types and getting editor support *just works*. No need for configurations, project setups, or any other tools.

It wasn't until lesson 6 that we eventually had to install something: external library type declarations that help us work with jQuery. This is because VS Code makes a lot of assumptions about your project structure and the files you work with. It's perfectly aligned to get you started easily.

And that's understandable. The VS Code team started at the same time as the TypeScript team, with common goals.[12] VS Code was designed to be the perfect TypeScript editor.

tsc

But what if you need TypeScript outside of VS Code? Because your co-workers use Sublime Text, or Atom, or VIM, or something entirely different? The tooling part of Type-

12 https://smashed.by/goto2016

Script is also available outside of VS Code. We can install the TypeScript command-line tool globally on our machine:

```
npm install -g typescript
```

With that, we get a tool called tsc, the TypeScript compiler. The TypeScript compiler's primary task is to take TypeScript code and compile it down to regular JavaScript. But with tsc, we can also type-check our JavaScript programs outside of any editor.

As with any compiler, there are lots of flags and configurations to be set. Let's go into our project's root folder. We create a folder called *@types* and move *ambient.d.ts* and *types.d.ts* into it. Then we run the following command in our terminal:

```
tsc --init
```

After we initialize a new project, we get a prefilled *tsconfig.json*. This is the main configuration file for TypeScript. Editors pick it up and behave accordingly, and when you run tsc in your terminal, TypeScript also refers to this file.

tsconfig.json

The prefilled values in *tsconfig.json* are pretty good when you want to start with a TypeScript project. We, however, want to type-check regular JavaScript, so we need to make a couple of adjustments.

This is how your *tsconfig.json* should look:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "es2020",
    "allowJs": true,
    "checkJs": true,
    "typeRoots": [
       "@types",
       "node_modules/@types"
    ],
    "esModuleInterop": true,
  }
}
```

There are a few settings and flags that are preset and are very useful once we compile TypeScript code to JavaScript code:

1. `target`: Once we compile TypeScript down to JavaScript, we need to define a target language specification.

This can be one of any recent or current ECMAScript standards (the standard JavaScript is based on), such as `ES3` for ancient browsers, `ES5` for legacy runtimes, or everything from `ES2015` to this year's specification. If you always go for the latest and greatest, use `ESNEXT`.

2. `module`. Another compiler flag that is important once we work with modules. If we do imports and exports in TypeScript, how should they be treated by the target language we compile to? Is it `commonjs` (used by Node.js), or `es2020` (a modern browser's module system), or one of the many that are around?

3. `esModuleInterop`. Strongly connected to `module` above. If you want to mix modules from different module systems like ES Modules and CommonJS, you can set this flag to `true`, and TypeScript will take care of compatibility.

We might not need them right now, but we will use them later on. The other compiler options are specifically useful when type-checking JavaScript:

1. `allowJs`. This flag tells TypeScript to allow a reference to regular JavaScript files.

2. `checkJs`. This flag is similar to the `//@ts-check` comment we used earlier. It tells TypeScript to type-check JavaScript files.

3. `typeRoots`. Here we tell the TypeScript compiler what VS Code originally did: the folders where additional type information can be found. One is our local *@types* folder where we load ambient module declarations. The other is *node_modules/@types* where we get the jQuery types from.

The automatically created *tsconfig.json* file is very well documented. For every flag, we see default values and a comment next to it explaining what this flag is supposed to do. Also, when working with VS Code, we get nice auto-completion features if we want to set and change settings.

Now the TypeScript compiler is configured, it's time to type-check. Let's run the following command in our terminal:

```
tsc --noEmit
```

tsc will pick up settings from *tsconfig.json*. The parameter `--noEmit` tells TypeScript that we just want to check types and not create any output files.

If everything in our code is all right, tsc exits with no error and no warning. If you want to regularly type-check your

code during development, you can add a `watch` mode so TypeScript reruns type-checking every time you save a file.

```
tsc --noEmit --watch
```

And with that, we've set up proper tooling that allows us to type-check JavaScript code outside of an editor, which is a good foundation for all the TypeScript examples to come!

Recap

In this chapter we had a gentle introduction to TypeScript from a tooling perspective. Our main goal was to get red squiggly lines in our code every time something seemed fishy!

Using a TypeScript-powered editor like VS Code, we were able to activate more and more features step by step:

1. Adding `//@ts-check` gave us an initial idea of what was wrong with our code. TypeScript uses a concept called *type inference* to automatically detect types of constants, functions, and variables.
2. Using JSDoc comments we were able to create custom types and annotate types throughout our code to

give TypeScript an even better understanding of our program's intent.

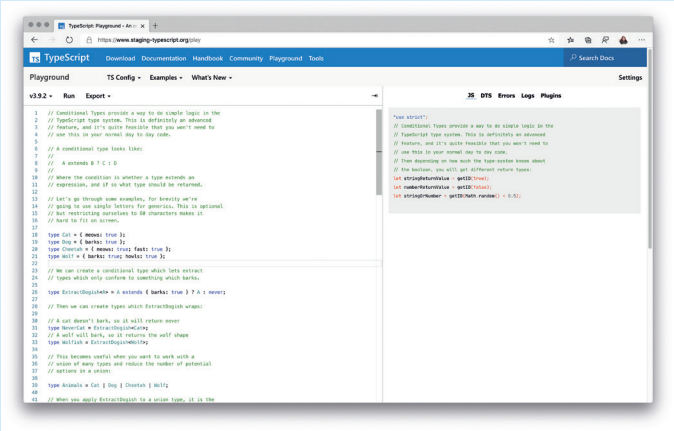
3. We heard that TypeScript is a *structural type system*, which means that TypeScript cares a lot about the shape or structure of objects and functions.
4. With *type declaration files* we wrote our first types in the TypeScript programming language, and referred to them in JSDoc comments.
5. *Ambient type declarations* allowed us to set type information for globals like jQuery or custom environment flags.
6. Last, but not least, we installed and configured the TypeScript compiler, so all implicit settings from the editor were written down and changeable. This allowed us to get the same results across editors and outside of editors. And it prepares us to do even more with TypeScript in the upcoming chapters.

Now that we've taken our first steps with TypeScript and are a little familiar with the way TypeScript affects our coding workflow, it's time to go all in with types in the next chapter!

Interlude: The TypeScript Playground

As you work more with TypeScript, you will start fiddling around in *tsconfig.json*, making sure your compiler setup is perfectly aligned with your project. Setting all these flags for your project sometimes makes it hard to go on type bug hunting. If a type just doesn't behave like you want it to, you can't say for sure if it's a library interfering, some ambient declaration files changing your types, or if the compiler version is too old.

Instead of fiddling around in your project setup, a safe bet is to head over to the TypeScript playground.



The TypeScript playground, v3.

The TypeScript playground is located at typescriptlang.org/play and offers an online IDE that allows you to experiment with types. You will find:

1. The most recent compiler versions, so you can find out if your type works in the version you actively use. This includes the nightly build, so you can experiment with upcoming features.
2. Lots of examples in both JavaScript and TypeScript so you can dig deeper into the possibilities that the type system has to offer.
3. An editor for *tsconfig.json*, allowing you to set all compiler flags and get information on possible values.
4. Example output when you run the code, or how it looks when it is compiled to JavaScript, or how it looks when you create a declaration file.
5. An export feature so you can continue in your projects, or you can file a nice issue on GitHub.

For me, the playground is the ideal place to experiment and work on more complex types before I add them to a project. The possibility of isolating declarations and types without any interference from the rest of the project is the ideal scenario for focus.

Chapter Two

WORKING WITH TYPES

Working with Types

We've learned that TypeScript is first and foremost a tooling layer: tooling that tries to understand our code even better than we do, pointing us to typos, errors, and possible pitfalls; tooling that lets us decide how much we want to commit to using it, and whose features can easily be activated when we want to get more information about our code.

Now that we are acquainted with TypeScript and feel more comfortable using it, it's time to dig a bit deeper into the eponymous types.

Lesson 8: Compiling TypeScript

Throughout this chapter, we are going to add a couple of bits and pieces that help us create an online shop: utility functions for the checkout, some dynamic overlays, and a little bit of back-end communication.

But this time we want to step out of our comfort zone and write actual TypeScript. And since TypeScript can't be executed by the browser, we have to compile it down to regular JavaScript.

Configuring the Compiler

To pick up from where we left off in our previous example, we are doing three things:

1. Rename our main JavaScript file to end with *.ts*. It's a TypeScript file now!
2. Create a new file, *example-two.ts* where we can add all examples from this chapter.
3. Adapt our *tsconfig.json* to ignore JavaScript.

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "es2020",
    "typeRoots": [
       "@types",
       "node_modules/@types"
    ],
    "esModuleInterop": true,
  }
}
```

The moment we run

```
tsc
```

in our terminal, we see that two new files pop up: a compiled JavaScript file for each TypeScript file. If you open them, you'll see that almost nothing has changed from our original TypeScript files. This is because we haven't done anything TypeScript-specific. Yet.

As TypeScript is a superset of JavaScript, all JavaScript code is TypeScript code. But since TypeScript is a superset, there's more to the language.

Our First Type Annotations

In *example-two.ts* we create a utility function for adding value added tax (VAT) to the regular price of a product. The price is provided in a currency of your choice; the VAT is a percentage expressed as a decimal (e.g. 20% is 0.2).

```
function addVAT(price, vat) {
  return price * (1 + vat)
}
```

TypeScript's type inference immediately kicks in. By looking at the calculation in the function's body, TypeScript knows we are dealing with numbers (the multiplication operator and the numeral 1 suggest that):

```
// vatPrice is of type 'number'
const vatPrice = addVAT(30, 0.2)
```

However, we are still allowed to pass anything but numbers as function parameters:

```
// vatPrice is of type 'number'
const vatPriceWrong = addVAT('this is so', 'wrong')
```

The value of `vatPriceWrong` is `NaN`, which stands for "not a number." Funnily enough, `NaN` is of type number, as it can only result from an operation on type number, but with one or many values not of type number.

Technically, then, TypeScript is correct. Our software is correct. But we want to be better than this. We want to make sure we don't get values we don't want to deal with. One way is to add a default value for `vat`:

```
function addVAT(price, vat = 0.2) {
  return price * (1 + vat)
}
```

Now TypeScript can infer again:

```
const vatPrice = addVAT(30, 0.2) // OK!
const vatPriceWithDefault = addVAT(30) // OK!
```

```
// Not OK. We expect a number for vat because of the
// default value! This piece causes errors
const vatPriceErrors = addVAT(30, 'a string!')

// This, however, is not quite reasonable, but OK
const vatPriceAlsoWrong = addVAT('Hi, friends!')
```

This is how far we can get with type inference. If we look at our function, `addVAT`, we can see:

- The return value is of type number, because of the kind of operations inside the function.
- `vat` is of type number, because the default value is a number.
- `price` is of type any.

> `any` is a special TypeScript type – it does not exist in JavaScript. It accepts any value of any type, and is thus a **top type**, encompassing all other types. TypeScript sets `any` as the default type for any value or parameter that is not explicitly typed or can't be inferred.

To be even more explicit and intentional with our types, we have to add type annotations. In chapter 1 we added type

annotations through JSDoc comments. Now we can do that directly in our function head, where it happens:

```
function addVAT(price: number, vat = 0.2) {
  return price * (1 + vat)
}
```

Notice that we annotated `price` to be of type number. We saw something similar in our *.d.ts* files from chapter 1.

With that, TypeScript will add beautiful red squiggly lines every time we pass a parameter that doesn't work.

```
const boom = addVAT('this is not a number!')
```

Instead of adding type annotations in comments, we add them directly to the names and parameters we declare. The type definitions above are short form for:

```
function addVAT(price: number, vat: number = 0.2):
  number {
  return price * (1 + vat)
}
```

Here we are even more explicit. We declare the types for both parameters, and even a return type (the last `number` before the curly brace).

Being so explicit has a special effect on TypeScript: TypeScript doesn't infer types anymore – it checks that the default value for `vat` and the return value of `price * (1 + vat)` match the types you declared in your function head.

Compiling to JavaScript

The browser's JavaScript runtime can't run TypeScript, though, so we have to get rid of all the annotations. Run tsc again in your terminal. TypeScript will take our new *example-two.ts* file, create an *example-two.js* file, and the contents will be the same, with all type annotations gone – JavaScript you can run in your browser.

> Note that after the compile step you lose all type safety when you run your code: it's plain JavaScript once it hits browsers. If somebody calls the `addVAT` function outside of your application, they can still run it with parameters of different types, potentially causing your application to break. If you expose your functions as an API to the outside world, it's always a good idea to do extra typeof checks and proper error handling.

You can also run `tsc` in watch mode to get regular updates once you save files:

```
tsc --watch
```

If you like, you can play around with some compiler options at this stage. Set `target` in your *tsconfig.json* to a different value (e.g. `es5`) and see what the compiled output looks like after you changed the settings. TypeScript not only erases all types but also takes modern-day JavaScript features and transpiles them to older ECMAScript versions.

TypeScript that generates JavaScript is called *emitting*. Remember the `--noEmit` flag from chapter 1? This allows us to check types without emitting any extra files. If this flag isn't set (the default) we always get JavaScript – even when type checks fail. If we want to make sure we don't get any emitted JavaScript output, set `noEmitOnError` to `true` in *tsconfig.json*.

Lesson 9: any, Are You OK?

In lesson 8, we made our first type annotations in functions. Type annotations can be of any primitive type, as well as composition types that exist in JavaScript. And there are more that are exclusive to TypeScript. Like `any`.

`any` is the default type if we neither specify a type nor let TypeScript infer one.

```
let deliveryAddress // deliveryAddress is any
```

The moment we assign a value, the type gets more specific:

```
// deliveryAddress is of type string
let deliveryAddress = '421 Smashing Hill, 90210'
```

Unless we explicitly specify the type through an annotation:

```
// deliveryAddress is of type any
let deliveryAddress: any = '421 Smashing Hill, 90210'
```

Of course, this only works if types are compatible. `any` can be anything: it's a top type and, therefore, all other types are part of `any`. If we assign an incompatible value to a variable of a certain type, TypeScript will throw squiggly lines at us:

```
// deliveryAddress is of type string,
// why assign a number?
let deliveryAddress: string = 2;
```

Left-Hand Typing vs. Right-Hand Typing

An explicit type annotation always goes first. The moment we annotate using the colon syntax, the name is of the type we annotate. All values have to follow and have to be compatible. We call this technique *left-hand typing*, as the typing happens before (to the left of) the equals sign.

Leaving out type annotations and working first with type inference is called *right-hand typing*: typing happens to the right of the equals sign, be it through a value assignment and inference, or via the type of a function's return value.

The same goes for composition types:

```
let deliveryAddresses = [
  '421 Smashing Hill, 90210',
  '221b Paw-ker Street',
  '4347 Whiskers-ia Lane',
]
// Type of deliveryAddresses is string[]
```

It's an array of strings, so the type is `string[]`. We can explicitly type `deliveryAddresses` to `string[]`.

```
let deliveryAddresses: string[] = []
```

```
// OK
deliveryAddresses.push('421 Smashing Hill, 90210')
// Not OK! 2 is not a string
deliveryAddresses.push(2000)
```

Both left-hand typing and right-hand typing have their pros and cons. With left-hand typing, you can think a lot about your types before you start coding the rest. Right-hand typing allows you to make up types as you go, which might be a little bit more JavaScript-y.

The Problem with any

Even though `any` is the base for all types within TypeScript and the default for everything TypeScript can't infer types from, you will rarely need to declare something as `any`. You usually want to have more information about your types rather than less.

With `any`, things like this are possible:

```
const myName: any = 'Fritz the Cat'
myName.firstLetter.makeCapitals()
```

Of course, properties like `firstLetter` and functions like `makeCapitals` don't exist in regular JavaScript types. But `any` doesn't know that, and doesn't care. For `any`, all contracts are fulfilled:

1. `myName`'s type is `any`, thanks to left-hand type annotation. `'Fritz the cat'` is of type `string`, but assignable to `any`. So this assignment is OK!
2. Since `any` can be anything, `any` also allows us to access properties that might not be there. They could be there – it can be anything, after all! This is, of course, utter nonsense, but that's what you get when you work with `any`.

`any` is a wildcard! Use `any` and you can go all out and forget about any type-checking at all. So why does something like `any` even exist?

This is due to the nature of JavaScript. In JavaScript, you are not bound to a type, and values from any type can appear in your variables and properties. Some developers make excessive use of that!

`any` reflects JavaScript's overarching flexibility; you can see it as a backdoor to a world where you want neither tooling nor type safety. By all means use `any`, but understand how it works and what to do with it – use it at your own risk!

Be certain that you want to use `any` explicitly as a type annotation. And if you want to enter through the backdoor to JavaScript flexibility, be very intentional through a type cast:

```
// theObject is an object we don't have a type for,
// but we know exactly what
// we are doing!
(theObject as any).firstLetter.toUppercase()
```

Of course, type casts also work with other types. If you want to make sure you don't have `any` somewhere in your code you don't expect it to be, set the compiler flag `noImplicitAny` to `true`. TypeScript will then make sure that you either assign values to correctly infer types or, in the case of `any`, make sure that you explicitly annotate or cast to `any`.

```
// deliveryAddress is of type any, because we
// didn't annotate a specific type. Implicit anys are
// hard to track down later on, that's why it's good
// to have TypeScript scream at us
function printAddress(deliveryAddress) {
  console.log(deliveryAddress)
}
```

If we annotate function parameters and variables explicitly with `any`, they become easier to track down later on once we have decided on the real types.

Lesson 10: Control flow

`any` is useful if you don't know which types to expect. The following function selects a delivery address, either one that has been passed as a parameter (of type `string`), or one from the `deliveryAddress` string array.

```
function selectDeliveryAddress(addressOrIndex: any) {
  if(typeof addressOrIndex === 'number') {
    return deliveryAddresses[addressOrIndex]
  }
  return addressOrIndex
}
```

A couple of things are happening here.

Type Narrowing

With `if(typeof addressOrIndex === 'number')`, we do something that connects the world of types with our JavaScript code.

Since JavaScript has numbers (see chapter 1), we can do several runtime type checks to make sure that a certain type is given. This has nothing to do with TypeScript, but does with JavaScript.

TypeScript, however, can make use of it. From this point on, TypeScript knows that `addressOrIndex` is of type number. So from this point on, we can access all the features of `number`.

We can format the number to a fixed-point representation:

```
if(typeof addressOrIndex === 'number') {
  // OK, because addressOrIndex is a number
  console.log(addressOrIndex.toFixed(2))
}
```

We can do all number operations on `addressOrIndex`:

```
if(typeof addressOrIndex === 'number') {
  // OK, because addressOrIndex is a number
  console.log(addressOrIndex * 2 + 3)
}
```

Or, in our case, we can use it as a number-based index for our array. We should check if it's within the range of the array, though:

```
// The comparison to see if addressOrIndex is
// smaller than the number
// of items in deliveryAddresses is also only
// possible because we know
// addressOrIndex is a number
```

```
if(typeof addressOrIndex === 'number' &&
   addressOrIndex < deliveryAddresses.length) {
  return deliveryAddresses[addressOrIndex]
}
```

We see a couple of TypeScript concepts in these few lines of code:

1. **Type guards.** Type guards perform run-time checks on types, just like the `typeof` operator makes sure we're dealing with a number at this point.
2. **Control flow analysis.** Type guards are used to trigger control flow analysis in TypeScript. TypeScript can analyze the flow of your program to provide the right types for the next steps.
3. **Narrowing down.** From the all-encompassing `any` type, we narrow down to the type `number`.

All three concepts are connected and are crucial to everything you are going to do with TypeScript from this point on.

Still any

Yet just as important as the things we do see are the things we don't. After narrowing `addressOrIndex`'s type down to

`number` with our *type guard*, we made it clear that we are expecting it to be a number at this point.

But what happens outside the if statement? What's the type of `addressOrIndex`? What's any without `number`? It's `any`! `any` is the all-encompassing top type. It can be anything, and everything is allowed.

So even after we make sure that we deal with `number` at a different point, the rest is still quite a lot!

And `any` in this position is very fragile. We expect `addressOrIndex` to be either `string` or `number`, but `any` allows us to pass anything and return everything, even if we explicitly type the return value:

```
function selectDeliveryAddress(addressOrIndex: any):
string {
  if(typeof addressOrIndex === 'number' &&
     addressOrIndex < deliveryAddresses.length) {
    return deliveryAddresses[addressOrIndex]
  }
// Totally OK with any
  return addressOrIndex
}

// Oh no! This is totally OK in TypeScript, but
// myFavouriteAddress is now string, even though we just
// return true? This is going to blow up in runtime!
const myFavouriteAddress = selectDeliveryAddress(true)
```

With the flexibility of `any` comes fragility and a huge potential for type mismatches. That's why we should avoid `any` at all costs.

Subtypes and Supertypes

Throughout the book we mention **subtypes** and **supertypes**. All types in TypeScript take their place in a hierarchy. For example, `any` is the supertype of all types, and `string` is a subtype of `any`.

Every value of `string` can be assigned to its supertype `any`, but not every value of `any` can be assigned to its subtype `string`.

The same concept applies to classes and objects. `HTMLElement` is the supertype of all HTML elements in the DOM. `HTMLAnchorElement` is a subtype of `HTMLElement`. Every `HTMLAnchorElement` can be assigned to type `HTMLElement`, but not every value of `HTMLElement` can be assigned to an `HTMLAnchorElement`.

With type narrowing we go down the hierarchy of types from supertypes to subtypes. In chapter 4 we'll see the full potential of Typescript subtypes.

Enter unknown

Thankfully, TypeScript has a partner to any: `unknown`. `unknown` is also compatible with every type in TypeScript, so it's also a top type.

But it's very inhibiting as well. Where we are allowed to do everything with `any`, we aren't allowed to do anything with `unknown`.

`unknown` should make you cautious: we have to provide a proper control flow to ensure type safety. Let's see what happens when we change `any` to `unknown`:

```
function selectDeliveryAddress(addressOrIndex:
unknown): string {

  if(typeof addressOrIndex === 'number' &&
     addressOrIndex < deliveryAddresses.length) {
    return deliveryAddresses[addressOrIndex]
  }
  return addressOrIndex
}
```

Boom! This is exactly what we want: "Type number is not assignable to type string." We must do type checks and trigger control flow analysis; otherwise, TypeScript will throw an error!

```
function selectDeliveryAddresses(addressOrIndex:
unknown): string {
  if(typeof addressOrIndex === 'number' &&
     addressOrIndex < deliveryAddresses.length) {
    return deliveryAddresses[addressOrIndex]
  } else if(typeof addressOrIndex === 'string') {
    return addressOrIndex
  }
  return ''
}
```

The control flow is complete. If we get a value of one of the two possible types, `number` or `string`, we know what to do: either return the entry from the list of delivery addresses, or return the delivery address we just entered. Should we pass anything else, we return an empty string!

Lesson 11: Typing Objects

In the previous examples, we worked a lot with primitive types: strings, numbers, Booleans. We also learned about two primitive top types that are not available in JavaScript but exclusive to TypeScript and TypeScript's type system: `any` and `unknown`.

`any` is both carefree and careless, putting type safety into the developer's hands; `unknown` requires much more caution and concern.

Composite Types

Both `any` and `unknown` are top types that include the whole set of other types that exist in JavaScript – like **composite types**.

Composite types are interesting. They can be virtually any combination of property names and other types, both primitive types and additional composite types. This makes the total space of possible types virtually endless.

Objects are composite types. Take this article from our online shop, for example:

```
const book = {
  title: 'Form Design Patterns by Adam Silver',
  price: 32.77,
  vat: 0.19,
  stock: 1000,
  description: 'A practical book on accessibility and
forms'
}
```

To define a type for this object, we can use the type alias syntax:

```
type Article = {
  title: string,
  price: number,
  vat: number,
  stock: number,
  description: string
}
```

With that, we just described the shape of the book object we created earlier. The same principles regarding left-hand typing and right-hand typing apply, as with primitives:

```
const movie: Article = {
  title: 'Helvetica',
  price: 6.66,
  vat: 0.19,
  stock: 1000,
  description: '90 minutes of gushing about Helvetica'
}
```

Here, we annotate the type of `movie`, which gets type-checked when we assign a value. Not having the correct properties, or missing properties altogether, would cause it to break:

```
// Property 'description' is missing
const movie: Article = {
```

```
  title: 'Helvetica',
  price: 6.66,
  vat: 0.19,
  stock: 1000,
}
```

Structural Typing and Excess Property Checks

If we assign a value with properties not in the specified type, TypeScript displays an error:

```
// Property 'rating' is not allowed
const movie: Article = {
  title: 'Helvetica',
  price: 6.66,
  vat: 0.19,
  stock: 1000,
  description: '90 minutes of gushing about Helvetica',
  rating: 5
}
```

However, this isn't the case when we define the value elsewhere:

```
// Property 'rating' is not allowed
const movBackup =  {
  title: 'Helvetica',
  price: 6.66,
```

```
  vat: 0.19,
  stock: 1000,
  description: '90 minutes of gushing about Helvetica',
  rating: 5
}

const movie: Article = movBackup // Totally OK!
```

Why? TypeScript is a *structural type system*. This means that as long as the defined properties of a type are available in an object, the structural contract is fulfilled. An apple has the shape of a sphere. Not a perfect sphere. There's the stalk and there are bumps all over, but it's still a sphere.

When assigning `movBackup` to `movie` of type `Article`, all relevant properties match: `title`, `price`, `vat`, `stock`, and `description`. The extraneous – or excess – `rating` property is swept under the rug. Literally! If we look at the autocompletion features that VS Code gives us as soon as we assigned `movBackup` to `movie`, we see that `rating` is not available anymore:

The structural contract is fulfilled. All excess properties are not available anymore.

This doesn't mean that these properties aren't there at runtime. They are! But during development, our tooling environment will make sure we only use the properties that are defined by the type.

TypeScript is very kind to us. We could get values from anywhere, and those values could change over time, but our contract still only cares about the right types of a certain set of properties. This makes our application still valid and type-safe but allows us to be flexible in other parts of our app.

This is also true if we have two different types with a similar enough structure to fulfill the contract:

```
type ShopItem = {
  title: string,
  price: number,
  vat: number,
  stock: number,
  description: string,
  rating: number
}

const shopitem = {
  title: 'Helvetica',
  price: 6.66,
  vat: 0.19,
  stock: 1000,
  description: '90 minutes of gushing about Helvetica',
  rating: 5
```

```
}

const movie: Article = shopitem // Totally OK!
```

But why does a direct value assignment after a type annotation cause an error?

```
// Property 'rating' is not allowed
const movie: Article = {
  title: 'Helvetica',
  price: 6.66,
  vat: 0.19,
  stock: 1000,
  description: '90 minutes of gushing about
Helvetica',
  rating: 5
}
```

This feature is called an *excess property check*. Because TypeScript is kind to us as structures could change across our application, it will point us to things that might be deliberate mistakes.

Assigning a value right after a type annotation that doesn't completely match is most likely an unintentional error. Why would we annotate a specific type and then assign something different?

Of course, having too few properties in our value causes an error in any case:

```
const missingProperties = {
  title: 'Helvetica',
  price: 6.66
}

// Boom! This breaks
const anotherMovie: Article = missingProperties
```

The structural contract is not fulfilled.

Objects as Parameters

We can also use our custom defined types as parameters in functions:

```
function createArticleElement(article: Article):
string {
  const title = article.title
  const price = addVAT(article.price, article.vat)
  return `<h2>Buy ${title} for ${price}</h2>`
}
```

And we can pass parameters with no explicit type annotation. Since the structural contract is fulfilled, TypeScript will be happy:

```
const shopitem = {
  title: 'Helvetica',
  price: 6.66,
  vat: 0.19,
  stock: 1000,
  description: '90 minutes of gushing about
Helvetica',
  rating: 5
}

createArticleElement(shopItem) // Totally OK!
```

This also means we can be very intentional with the type we want for the `createArticleElement` function, and maybe do an inline object type with only the properties we expect:

```
function createArticleElement(
  article: { title: string, price: number, vat: number
}): string {
  const title = article.title
  const price = addVAT(article.price, article.vat)
  return `<h2>Buy ${title} for ${price}</h2>`
}
```

Passing elements of type `Article` still would work:

```
const movie: Article = {
  title: 'Helvetica',
  price: 6.66,
  vat: 0.19,
  stock: 1000,
  description: '90 minutes of gushing about Helvetica'
}
createArticleElement(movie)
```

The structural contract is still fulfilled.

But just like it is with direct value assignments, passing an object with too many properties directly to a function will trigger excess property checks:

```
createArticleElement({
  title: 'Design Systems by Alla Kholmatova',
  price: 20,
  vat: 0.19,
  rating: 5
}) // Boom! rating is one property too many
```

Structural typing is very different from other programming languages, but it suits JavaScript and how we work with JavaScript so well. Passing around data that evolves –

getting more properties than before or maybe even losing some properties – is very common. All we should care about is that all properties that we need are available. This is the contract for our objects and functions.

Lesson 12: Object Type Tool Belt

When writing TypeScript, we work a lot with object types. Almost everything in JavaScript is either a function or an object, so it makes sense to spend some time with them! In this lesson, we'll look at a couple of extras that might help you when working with object types.

typeof

Object types can be very long. Sometimes we work with data structures that are deeply nested and have tons of properties. Look at the object that defines a default order in our online shop:

```
const defaultOrder = {
  articles: [
    {
      price: 1200.50,
      vat: 0.2,
      title: 'Macbook Air Refurbished - 2013'
```

```
    },
    {
      price: 9,
      vat: 0,
      title: 'I feel smashing subscription'
    }
  ],
  customer: {
    name: 'Fritz Furball',
    address: {
      city: 'Smashing Hill',
      zip: '90210',
      street: 'Whisker-ia Lane',
      number: '1337'
    },
    dateOfBirth: new Date(2006, 9, 1)
  }
}
```

This object is a bit complex! We could define the type in one sitting:

```
type Order = {
  articles: {
    price: number,
    vat: number,
    title: number
  }[],
  customer: {
    name: string,
    address: {
```

```
      city: string,
      zip: string,
      street: string,
      number: string
    },
    dateOfBirth: Date
  }
}
```

Or we could create lots of smaller types:

```
type ArticleStub = {
  price: number,
  vat: number,
  title: string
}

type Address = {
  city: string,
  zip: string,
  street: string,
  number: string,
}

type Customer = {
  name: string,
  address: Address,
  dateOfBirth: date
}

type Order = {
  articles: ArticleStub[],
```

```
  customer: Customer
}
```

Or a mix of both. In either case, we end up either maintaining a lot of types or creating unwieldy types. All we wanted was to get a quick type for a data structure in order to have better auto-completion and type safety in our methods.

Remember the `typeof` operator we met in lesson 10? With the `typeof` operator, we were able to do type checking during runtime. In TypeScript's type system, the `typeof` operator takes any object (or function, or constant) and extracts the *shape* of it:

```
type Order = typeof defaultOrder
```

This gives us a type we can use anywhere in our code:

```
/**
 * Checks if all our orders have articles
 */
function checkOrders(orders: Order[]) {
  let valid = true;
  for(let order of orders) {
    valid = valid && order.articles.length > 0
  }
  return valid
}
```

The moment you update your `defaultOrder` object, the type `Order` gets updated as well!

Optional Properties

In the previous example, we used a form of `Article` that misses two properties we originally defined: `stock` and `description`. That's already the second time we had no use for both properties. Remember `createArticleElement`?

It looks like we need a simple `Article` type more often than we think. So what should we do? Create two types, `Article` and `ArticleStub`? Set dummy values for properties that are not necessary? Set the properties deliberately to `undefined`?

Any of these sounds a little fishy, and not very JavaScript-like. Like ways to satisfy the type system that only generate more code. And TypeScript shouldn't be like that. It shouldn't get in your way. It shouldn't force you to care about types – you should care because you want to care.

The best way would be to adapt the original `Article` type and set two optional properties:

```
type Article = {
```

```
  title: string,
  price: number,
  vat: number,
  stock?: number,
  description?: string
}
```

A question mark after a property's name declares that property optional. What does this mean when we code? Well, optional parameters are... optional. This means they can be available, but they could also be missing. We have to check if they are available:

```
function isArticleInStock(article: Article) {
  // this check is necessary to make sure
  // the optional property exists
  if(article.stock) {
    return article.stock > 0
  }
  return false
}
```

Our type becomes much more flexible and can be used in many more scenarios.

Exporting and Importing Types

Now that our `Article` type is so flexible, we want to share it with other parts of our application. When working with JSDoc comments, we imported types on occasion. We can do the same thing when writing pure TypeScript.

First, we make the type available by *exporting* it:

```
export type Article = {
  title: string,
  price: number,
  vat: number,
  stock?: number,
  description?: string
}
```

Then, we import `Article` using the same function we used in chapter 1. But this time we import types via regular `ECMAScript` imports:

```
import { Article } from './example-two'

const book: Article = {
  price: 29,
  vat: 0.2,
  title: 'Another book by Smashing Books'
}
```

Like all type annotations, this is erased when we compile. The same import syntax is used to import other elements (objects, functions) from the *example-two* file as well. If we are only interested in types, we use a slight variation on the regular ECMAScript import: a type import.

```
import type { Article } from './example-two'

const book: Article = {
  price: 29,
  vat: 0.2,
  title: 'Another book by Smashing Books'
}
```

This is especially helpful when you deal with lots of imports from a particular file – both types and other elements – and want to separate type information from the rest of the application.

Lesson 13: Typing Classes

TypeScript can be seen as a programming language that *erases* to JavaScript. On top of JavaScript code that exists and is valid, we add another layer of type information that disappears once the compiler has run.

We can distinguish between declarations that create types and elements that create values. Types are only available in TypeScript. They add a new name to an existing type space, we use them for tooling, and they disappear the moment we compile down to JavaScript.

Value-creating declarations still exist in JavaScript. Functions, variables – stuff that remains after compilation. Separating these two worlds helps greatly when working with TypeScript, as you can strip away the type-creating declarations and still have proper JavaScript that almost looks the same.

There is one thing, though, that contributes to both the type-creating space and the value-creating space: classes.

Classes in JavaScript

Since the ECMAScript 2015 standard, JavaScript features classes as an alternate syntactic form for the constructor function and prototype pattern. Here's a class in pure JavaScript that applies a discount to one of our articles:

```
class Discount {
  isPercentage
  amount
```

```
  constructor(isPercentage, amount) {
    this.isPercentage = isPercentage
    this.amount = amount
  }

  apply(article) {
    if(this.isPercentage) {
      article.price = article.price
        - (article.price * this.amount)
    } else {
      article.price = article.price - this.amount
    }
  }
}

// A discount that shaves off 10 EUR
const discount = new Discount(false, 10)
discount.apply({
  price: 39,
  vat: 0.2,
  title: 'Form Design Patterns'
})
```

With a few little extra type annotations, we can have proper tooling and can make sure that we construct correct objects:

```
class Discount {
  isPercentage: boolean
  amount: number
```

```
  constructor(
    isPercentage: boolean,
    amount: number) {
    this.isPercentage = isPercentage
    this.amount = amount
  }

  apply(article: Article) {
    if(this.isPercentage) {
      article.price = article.price
        - (article.price * this.amount)
    } else {
      article.price = article.price - this.amount
    }
  }
}
```

The moment we create a class, it's available in the type space as well:

```
let discount: Discount
  = new Discount(true, 0.2)
```

With custom object types, we always describe the shape of an object and make sure that all values passed as parameters or assigned to variables match that shape.

Structural Typing with Classes

Since TypeScript is a structural type system, we are also more interested in the shape of the objects that are created by a class, rather than the class itself. So what's the shape of a class-generated object?

Classes have two parts:

1. **The constructor function**. We defined our constructor function to take a Boolean `isPercentage`, and a number for the amount we want to shave off. The moment we call `new Discount(true, 0.2)`, we invoke the constructor function.
2. The second part is a **prototype**. This is everything around the constructor function: two fields (`isPercentage, amount`) and a function to apply the discount to an article.

The prototype defines the shape of the object that is returned by invoking the constructor. Now that we know the shape, we can even assign regularly generated objects to a variable of type `Discount`.

```
let allProductsTwentyBucks: Discount = {
  isPercentage: false,
```

```
  amount: 20,
  apply(article) {
    article.price = 20
  }
}
```

This is a valid `Discount`, as the shape is intact. But it changes the semantics of the `Discount` class tremendously.

This also works vice versa. We can define an object type, and create a new `Discount` object via a constructor:

```
type DiscountType = {
  isPercentage: boolean,
  amount: number,
  apply(article: Article): void
}

let disco: DiscountType = new Discount(true, 0.2)
```

In a structural type system, only the shape is important. Names are sound and smoke.

Extending Classes

One main feature of classes is that they are extensible. You can take an existing class and extend it, overriding and adding features.

```
/**
 * This class always gives 20 %, but only if
 * the price is not higher than 40 EUR
 */
class TwentyPercentDiscount extends Discount {
  // No special constructor
  constructor() {
    // But we call the super constructor of
    // Discount
    super(true, 0.2)
  }

  apply(article: Article) {
    if(article.price <= 40) {
      super.apply(article)
    }
  }
}
```

We created a discount class that always applies 20%, but only if the article's price is lower than 40 Euro. In this

special case, `TwentyPercentDiscount` is of the same shape as `Discount`, which means that their type declaration is interchangeable:

```
let disco1: Discount
  = new TwentyPercentDiscount() // OK
let disco2: TwentyPercentDiscount
  = new Discount(true, 0.3) // OK! Semantics changed!
```

But we can change the shape. Let's create a validation feature to `TwentyPercentDiscount`:

```
class TwentyPercentDiscount extends Discount {
  constructor() {
    super(true, 0.2)
  }

  apply(article: Article) {
    if(this.isValidForDiscount(article)) {
      super.apply(article)
    }
  }

  isValidForDiscount(article: Article) {
    return article.price <= 40
  }
}
```

The shape has changed, which means that the same rules as for object types apply: if more properties are available, the shape contract is satisfied; if properties are missing, the shape contract is not fulfilled:

```
let disco1: Discount
  = new TwentyPercentDiscount() // Still OK!

// Error! We miss the `isValidForDiscount`
// method
let disco2: TwentyPercentDiscount
  = new Discount(true, 0.3)
```

By now, classes have become a mainstay in JavaScript, especially since component-based frameworks rely heavily on classes to define components. Typing classes might be a little bit confusing, as it merges the two worlds of type creation and value creation, but as long as we keep the main principles of a structural type system in mind, they're as easy to use.

And they'd better be. Classes were one of TypeScript's first killer features that brought people from the C# and Java worlds to JavaScript. TypeScript featured one of the first ECMAScript class implementations as the first proof of concept that classes could work in JavaScript. The syntax hasn't changed much since then.

Lesson 14: Interfaces

When working with types, you will come across interfaces at some point. When classes were introduced to TypeScript, interfaces followed along, carried over from object-oriented programming languages that were popular at that time.

Historically, classes and interfaces form a strong relationship, with interfaces describing the blueprint of a class: structural information that has to be implemented by the respective class. This is where the `implements` keyword is introduced.

This sounds an awful lot like the relationship between custom object types and objects. And in fact, as TypeScript evolved, interfaces became pretty much indistinguishable from custom object types. There are still a few subtle differences, though.

Describing Interfaces

Let's say we work with another team that works on a different part of the same online shop. They write their own code but use the same data structures. And they prefer interfaces.

An `Article` type in our codebase becomes a `ShopItem` interface in their codebase.

```
// Our Article type
type Article = {
  title: string,
  price: number,
  vat: number,
  stock?: number,
  description?: string
}

// Our friend's ShopItem
interface ShopItem {
  title: string;
  price: number;
  vat: number;
  stock?: number;
  description?: string;
} // And yes, the semicolons are optional
```

The syntactic differences are so subtle, you'd be forgiven for hardly noticing them. Of course, both `Article` and `ShopItem` are compatible, because their shape – their structure – is the same:

```
const discount = new Discount(true, 0.2)
const shopItem: ShopItem = {
  title: 'Inclusive components',
  price: 30,
  vat: 0.2
}
```

```
// Discount.apply is typed to take `Article`
// but also takes a `ShopItem`
discount.apply(shopItem)
```

If you use classes, both interfaces and types can be *implemented*:

```
// Implementing Interfaces
class DVD implements ShopItem {
  title: string
  price: number
  vat: number
  constructor(title: string) {
    this.title = title
    this.price = 9.99
    this.vat = 0.2
  }
}
// Implementing Types
class Book implements Article {
  title: string
  price: number
  vat: number

  constructor(title: string) {
    this.title = title
    this.price = 39
    this.vat = 0.2
  }
}
```

With that, we make sure that the class we're creating adheres to the shape we want. If we miss any properties that are required, TypeScript alerts us:

```
// Nope, we missed the property `title`!
class Book implements Article {
  price: number
  vat: number

  constructor() {
    this.price = 39
    this.vat = 0.2
  }
}
```

Of course, the shape of objects of types `Book` and `DVD` are the same as `Article` or `ShopItem`, so our `Discount` class works on them as well:

```
let book = new Book('Art Direction on the Web')
discount.apply(book)

let dvd = new DVD('Contagion')
discount.apply(dvd)
```

Discounts everywhere!

Declaration Merging

At first glance, interfaces and types seem to be entirely the same. Remember that for later, when we see advanced techniques with object types. You can substitute interfaces for object types at any time. Other than historically, where are the differences? Aside from some nuances, the biggest difference is **declaration merging**.

Declaration merging for interfaces means we can declare an interface at separate positions in the same file, with different properties, and TypeScript combines all declarations and merges them into one.

We can take our `ShopItem` declaration from earlier on, and extend them with an array of `reviews` at a totally different position:

```
interface ShopItem {
  reviews: {
    rating: number,
    content: string
  }[]
}
```

Adding this couple of lines will break all usage of `ShopItem` throughout our file, as the `reviews` property is not optional, and `DVD` and other elements will have to implement it.

Declaration merging in one file, while possible, may seem a little nonsensical. Isn't it more readable and understandable if we have all properties in one declaration? Of course it is!

But there's a special use case where declaration merging makes a lot of sense. Think back to chapter 1, when we defined a global variable in an ambient type definition file.

When writing JavaScript, we often get into situations where variables, functions, or classes from somewhere outside become available globally. Not only an `isDevelopment` flag, but also maybe an analytics script that allows you to get statistics on your site's usage. Or the YouTube API that allows you to include and play different YouTube movies.

All these things usually hang themselves into the global object. In browsers, that's the window object. The window object is very much defined through the `Window` interface.

Wouldn't it be great if we could extend `Window` from anywhere in our code, making global flags, APIs, and functions available anywhere? This is just a few lines' worth of code:

```
declare global {
  interface Window {
    isDevelopment: boolean
  }
}
```

First, we open the `global` namespace. Namespaces are a feature that came before the times of proper module encapsulation. They are mostly used when we want to spread type declarations across files, like all declarations that are globally available (`window`, `document`, `navigator`, and so on).

Namespace declarations can also be merged.

Next, we open the `Window` interface. We don't overwrite the entire type; we attach a custom field to it: `isDevelopment` of type `boolean`. With that declaration, anywhere in our code, we can immediately check if we are in development mode:

```
class Discount {
  ...
  apply(article: Article) {
    ...
    // Here we check if we are in dev mode
    if(window.isDevelopment) {
      console.log('Another discount applied')
    }
  }
}
```

The `Window` interface is usually very aware of the current state of browsers that support a certain compile `target` in your *tsconfig.json*. Which means it's possible there are fea-

tures available in your browser that don't yet have types in TypeScript, so it could throw errors.

This is because TypeScript goes for the lowest, common denominator in `Window`. If you want to use newer features that aren't available right away, and test accordingly if they do exist, you can use the technique above to add proper types. I've published an article on Fettblog (my website) that describes how '`ResizeObserver`' types have been added to Window,[14] and how to safely use them.

Recap

We've covered a lot in this chapter. Even though TypeScript is just a small layer on top of JavaScript, the type system itself is so manifold and flexible that we can do a ton of things with it. And this is just the start! Let's recap:

1. We learned how type annotations work and how we can erase to JavaScript using the TypeScript compiler. Turns out, TypeScript not only adds a type layer but compiles down to various versions of ECMAScript.
2. We saw that TypeScript comes with its own primitive types, like `any`: a wildcard that lets you go haywire with your type safety, but ensures you are not blocked by types when you know better.

14 https://smashed.by/resizeobserver

3. This makes TypeScript a *gradual* type system. Use types when you feel like it and when you see a benefit, not when a tool tells you to.
4. We learned about *type guards*, *control flow* and *narrowing down*. Since JavaScript functions can take arguments of any type, we can do run-time type checks with `typeof` and *infer* new types from it.
5. We learned how to type objects, and what the key aspects of a structural type system are.
6. We also got a ton of tools to make creating types a lot easier. Make them on the go when you write regular JavaScript.
7. We also delved into a couple of object-oriented features, like classes. Classes are different as they contribute to both the value-creation and type-creation spaces.
8. We ended the chapter learning about interfaces (the older sibling of object types, brought up differently but which, in the end, turned out almost indistinguishable from its younger counterpart). The only feature worth noticing is *declaration merging*, allowing us to extend interfaces when we see a need for it.

This is the pure baseline of working with types. The rest of the book will focus on getting the best types, the best editor experience, and the most robust yet flexible type safety for your projects.

Interlude: Borrowed Language

TypeScript has two major aspects:

1. Innovation in a type system that allows the maximum type-safety for all possible JavaScript scenarios.
2. Provide the first implementation of new JavaScript features that can be transpiled to older versions of ECMAScript. This means that no language innovation happens without the proper process in TC39,[15] the ECMAScript standards committee.

The second part was not always the case. Back when TypeScript was created, lots of modern-day JavaScript language features weren't available – not even on the roadmap! This was a different JavaScript back then, a lot clunkier and raw. Some would say beautiful.

Even if JavaScript has evolved a lot since then, in 2012 people thought they needed some language features like classes to make their code more structured and scalable.

This encouraged the designers of TypeScript to include quite a few features that come from the object-oriented programming model of classic languages like Java or C#. Some, like classes, appeared in JavaScript. Others will most likely never land in JavaScript.

15 https://tc39.es/

But even if they don't appear in JavaScript, they are still here. And you can use them. If you're sure. These features are not bad – not at all! But they have some gotchas you should be aware of.

Property Access Modifiers

The way we write classes in this book is the usual JavaScript way of writing classes. TypeScript classes can do a lot more, even modifying access to certain properties:

```
class Article {
  public title: string
  private price: number

  constructor(title: string, price: number) {
    this.title = title
    this.price = price
  }
}

const article
  = new Article('Smashing Book 6', 39)

console.log(article.price)
```

Access modifiers are useful if you follow object-oriented programming patterns, but this feature is only available in

TypeScript. Nothing prevents you from accessing properties declared `private` when executing JavaScript. Furthermore, JavaScript now has its own private access modifier, that puts a little fence in front of properties nobody outside should see:

```
class Article {
  #price: number

  constructor(price: number) {
    this.#price = price
  }
}
```

Along with new keywords and changed semantics comes also a lot of syntactic sugar that makes you more productive but deviates even further from JavaScript:

```
class Article {
  // This constructor sets properties
  // automatically

  constructor(private price: number) {
    // Nothing to do here, this.price is
    // still available
  }
}
```

If you do a lot of object-oriented programming, those features might be useful to you. But be aware that this is added syntax on top of JavaScript that will not find its way into the standard. If you don't want to wade in to added syntax, it might be a good idea to stick with standard JavaScript class features.

Abstract Classes

Also coming from the OOP world: abstract classes. When interfaces describe the blueprint of implementations, and classes are implementations, abstract classes are something in-between. They implement a lot but leave out important details to be filled out by real classes:

```
abstract class Discount {
  // This needs to be implemented
  abstract isValid(article: Article): boolean;

  // This is already implemented
  apply(article: Article) {
    // Like before ...
  }
}
```

TypeScript reports errors when you forget to implement `isValid`, and you're not allowed to create an object with an

abstract class. But again, only in TypeScript. Abstract classes are emitted to JavaScript, but their semantics are similar to regular classes. Only now, they're missing entire implementation details. This can cause weird errors at runtime that we can easily avoid.

Enums

Enums (short for enumerations) allow you to bundle a couple of types and use them throughout your code:

```
enum UserType {
  Admin,
  PayingCustomer,
  Trial
}

function showWarning(user: UserType) {
  switch(user) {
    case UserType.Admin:
      return false;
    case UserType.PayingCustomer:
      return false;
    case UserType.Trial:
      return false;
  }
}
```

They have one significant aspect: they contribute to both the type-creation space and the value-creation space, emitting code. Occasionally. Adding a `const` keyword before `enum UserType` prevents code emitting.

While this type information might be handy at times, they are not as safe as other structures (which we will see in later lessons), nor do they look anything like TypeScript's syntax in the emitted code. Also, forgetting the `const` keyword can result in lots of additional unnoticed lines of code in the final output, and with a significant increase in file size when shipped.

Use with care. If you want to know more, Axel Rauschmayer has written a lengthy post on all aspects of enums.[16]

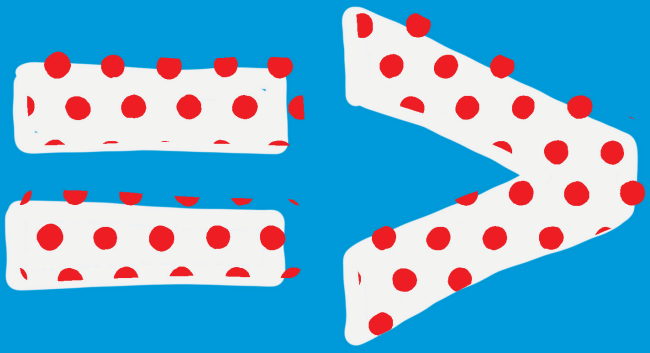
A Rule of Thumb

There is more, and the official TypeScript documentation[17] provides a lot of insight if you want to learn. But becoming an expert TypeScript developer doesn't require it. The magic nowadays lies in the type system and the mammoth task to make all of JavaScript understandable.

16 https://smashed.by/enums
17 https://typescriptlang.org

For us, this means that we can stick to the separation of concerns:

1. In terms of programming language, write JavaScript – write modern JavaScript. Let TypeScript make sure you can write modern JavaScript.
2. In terms of the type system, learn the type system. Find ways to annotate your functions so you get the best tooling and the best possible type safety. The rest of this book is going to help you with exactly that.



Chapter Three

FUNCTIONS

Functions

We want to get the best possible tooling for ourselves and our colleagues. If you don't currently share your code with other people, then your future self wants you to get the best possible tooling. Code you remember is good. Code that tells you what it means is better!

Now that we know the basics of TypeScript's types, we'll set the following goals for the rest of the book:

1. Write modern JavaScript!
2. Find the right types to make sure our JavaScript makes sense.

In chapter 3 we'll take a good look at functions. Functions are essential in JavaScript, and there are lots of different typing scenarios available to us. To make functions tangible, we'll look at a website's search field: a search field with type-ahead, that shows some results the moment a user types a search query.

For the following examples, it's recommended to add the `strict` flag to your *tsconfig.json* and set it to `true`. Even more red squiggly lines!

Lesson 15: A Search Function

Functions are everywhere in JavaScript. We saw a few functions in the last couple of examples. Functions contribute to the value-creation space of TypeScript.

Their parameters have types, which are any by default and can be inferred from default values. The return values have types; they can be inferred from the actual value that we return in the function body.

We are going to create a search field for our website. The moment we enter a query, we call a back end that provides us with results in JSON. In our program, we create a search function that takes query parameters, calls the back-end search API, and returns the correct results.

Typing Function Heads

Let's look at the function head of our search function. We use the `declare` keyword to make the function available without implementing a function body at the moment. This is a great way to think about types and the function's interface before going into the details. We'll remove the declare keyword later and make sure we get a proper implementation.

```
// A helper type with the results we expect
// from calling the back end
type Result = {
  title: string,
  url: string,
  abstract: string
}

/**
 * The search function takes a query it sends
 * to the back end, as well as a couple of tags
 * as a string array, to get filtered results
 */
declare function search(
  query: string,
  tags: string[]
): Result[]
```

We explicitly typed the parameters and return values of the function head, which is good practice as it lets you make sure you not only get the right return values and process them further, but you can also validate your implementation against the explicit types you expect. The function head has some issues:

1. Tag filters are required. You wouldn't be able to call the search function without any tags provided, even if it's just an empty array.

2. The search function is implicitly synchronous. We call search and expect an array of results in return. Calling a back end usually involves an asynchronous operation. Well, at least it should! In our case, we want the search function to work asynchronously.

Let's solve each issue.

Optional Parameters

Tags are nice to filter results based on preferences. Think of a website that offers articles on JavaScript, CSS, design, art direction, and UX. Can you think of any? The moment you search for "Ember," for example, you need to distinguish between the JavaScript framework and the digital scrapbook. That's why you select tags.

But tags are optional, at least in our case. Our function doesn't know yet:

```
search('Ember', ['JavaScript']) // Works
search('Ember') // Errors! Tags are missing
search('Ember', []) // Nasty workaround
```

Passing empty values doesn't seem very JavaScript-like. Just like objects, functions can take optional parameters, marked with a question mark:

```
declare function search(
  query: string,
  tags?: string[]
): Result[]
search('Ember') // Yes!
search('Ember', ['JavaScript']) // Also yes!
search('Ember', ['JavaScript', 'CSS']) // Yes yes!
```

Also, just like optional properties in objects, you have to check if they're available once you write your function body.

Asynchronous Back-End Calls

To make our function correctly asynchronous, we are going to work with promises. Promises are JavaScript's main construct to deal with asynchronous tasks. They're called promises because they promise to resolve to a value… eventually. We just don't know when. One of the most popular promise-based tasks is *fetch*, a handy way to call a back end and receive data.

Let's use fetch to implement our search function. Remove the `declare` keyword from the function head. We are not declaring a function anymore, we're implementing it. Also, for now, let's remove the return value type from our function declaration. We're going to add one later on.

```
function search(query: string, tags?: string[]) {
  // Based on our input parameters, we build a query
  // string
  let queryString = `?query=${query}`

  // tags can be undefined as it's optional.
  // let's check if they exist
  if(tags && tags.length) {
    // and add all tags in that array to the
    // query string
    queryString += `&tags=${tags.join()}`
  }

  // Ready? Fetch from our search API
  return fetch(`/search${queryString}`)
    // When we get a response, we call the
    // .json method to get the actual results
    .then(response => response.json())
}
```

A few things to notice here:

1. We have to check if `tags` exists. When you hover over the optional `tags` parameter in your function body, you can see that it can be a string array, but it can also be undefined. Only if we check the value exists will TypeScript allow us to use the array methods and append to the query string.

2. `fetch` returns a promise. Promises are then-able, which means that once the value is here, we can access the callback of the `then` function. It's interesting to watch types at this point. Hover over `fetch` to see that the `fetch` function returns a `Promise<Response>`, and the `response` param inside the callback is of type `Response`. If you ever see a type with `<` and `>` signs, remember the name *generic*. Promises can be of so many things. If we want to specify the type of the return value, we use a generic to set it to; for example, `Response`. We are going to see a lot of generics in the later chapters of the book.

One nice thing about TypeScript is that you don't need to know all APIs by heart. Once you call `fetch`, your editor gives you hints on what you can pass as arguments. When you are in the `then` callback, you don't have to know that `response` has a `.json()` function. You can browse through a list of suggestions from your editor and select the one you think is most appropriate. Most likely you will stop at `.json()` and think: "Ah yes, that's what I was looking for."

`fetch` has a return value of type `Promise<Response>`. The next then returns the return value of `response.json`, which also becomes the return value type of our search function. Type inference!

However, the type of `response.json`'s return value is `Promise<any>`. And rightfully so! How should TypeScript know what you get once you call your back end? What we want is actually what we get: a promise of results. Or, in types: `Promise<Result[]>`

Here we have to be explicit, either through a type cast:

```
function search(query: string, tags?: string[]) {
  ...
  return fetch(`/search${queryString}`)
    .then(response =>
      response.json() as Promise<Result[]>)
}
```

which is OK, as we are more explicit about the type at the position where we get any. Or the other possibility is the function head:

```
function search(
  query: string,
  tags?: string[]): Promise<Result[]> {
  ...
  return fetch(`/search${queryString}`)
    .then(response => response.json())
}
```

Both versions work the same: `any` is compatible with any other type. It's the wildcard, the happy-go-lucky type where anything can happen. We say explicitly that we expect a `Result` array instead of `any`. And TypeScript accepts that!

Which version you use is up to you. Type casts are quick and available where they are needed, but might be overlooked at times. I prefer explicit function heads and to keep the JavaScript inside the function body as it is.

Lesson 16: Callbacks

In the previous lesson, we dealt with types of return values and parameters in functions. But functions have types too! Let's have a look at our search function from the previous lesson:

```
function search(
  query: string,
  tags?: string[]
): Promise<Result[]> {
  let queryString = `?query=${query}`
  if(tags && tags.length) {
    queryString += `&tags=${tags.join()}`
  }
  return fetch(`/search${queryString}`)
    .then(response => response.json())
}
```

We can get the function's type most easily by calling `typeof`:

```
type SearchFn = typeof search
```

When you hover over `SearchFn`, you'll see the expanded type definition for a function:

```
type SearchFn = (
  query: string, tags?: string[] | undefined
) => Promise<Result[]>
```

This is very similar to JavaScript's arrow notation for functions. We see: the opening of a function's head, declaring the parameter query, which is a string; `tags`, which can be a string array or undefined since it's optional; then comes the arrow, and the function returns a promise of results.

Of course, we can type the function type as defined up there, without using `typeof`. But `typeof` is handy!

Function Types in Objects

So what can we do with function types? Quite a lot! We can define complex object types that contain functions:

```
type Query = {
  query: string,
  tags?: string[],
  assemble: (includeTags: boolean) => string
}
```

This defines a query object that not only has the query itself but optional tags and a function that assembles the query string for a possible search. An object like this would satisfy the contract:

```
const query: Query = {
  query: 'Ember',
  tags: ['javascript'],
  assemble(includeTags = false) {
    let query = `?query=${this.query}`
    if(includeTags && typeof this.tags !==
'undefined') {
      query += `&${this.tags.join(',')}`
    }
    return query
  }
}
```

And types are composable, so we can define the function type for `assemble` at a different position:

```
type AssembleFn = (includeTags: boolean) => string
type Query = {
  query: string,
  tags?: string[],
  assemble: AssembleFn
}
```

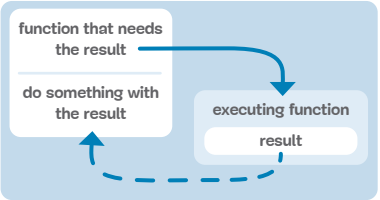
This behaves just like JavaScript, where we can also define functions outside of the main object.

Function Types in Functions

Let's say we want to have our search function be pluggable, and want to create different search functions for different scenarios. Maybe some that work differently, but always listen to the same parameters: a query, a list of tags. And they return a promise with results. We already defined the search function like this:

```
type SearchFn = (
  query: string, tags?: string[] | undefined
) => Promise<Result[]>
```

We can use this function type to create different functions that follow the same signature.



Callbacks are pluggable functions. For any task that can't return a result directly – maybe due to asynchronous execution – we can pass a function that takes the result and works with it.

This is incredibly helpful for callback functions. Functions are first-class citizens in JavaScript and can be used as values, just like any string or number. So we can pass functions as arguments to another function. Usually, this type of function is called a *callback function*, as they should call back to the spot where the function has been invoked. Let's write a function that combines a few workflows for our search:

1. selecting the element where the user inputs their query
2. calling a search
3. showing the results

It takes three arguments:

1. the ID of the input element
2. the ID of the element to present the results in
3. a search function

The types of the two parameters are pretty much straightforward. Both IDs are strings. The search function uses said function type.

Again, think first about the function head before developing the body. Let's write a function head that takes two strings for the IDs of our elements, as well as a search function:

```
declare function displaySearch(
  inputId: string,
  outputId: string,
  search: SearchFn
): void
```

Of course, we could type the function type explicitly. But sometimes it's much more readable to have a separate type for that. We use `void` as the return value as we don't expect anything to return.

Anonymous Functions

Later on we'll deal with the meaning of `void` (lesson 17) and the implementation of `displaySearch` (lesson 18). For now, let's focus on the possible search functions we can pass; for example, the original search function:

```
displaySearch('searchField', 'result', search)
```

Or an entirely new one that we just made up. Look at this test function:

```
displaySearch(
  'searchField',
  'result',
  function(query, tags) {
    return Promise.resolve([{
      title: `The ${query} test book`,
      url: `/${query}-design-patterns`,
      abstract: `A practical book on ${query}`
    }])
  }
)
```

We notice a few things here:

1. The function is anonymous: it has no name. It has just been passed as an argument to `displaySearch`.
2. But it still follows the contract of `SearchFn`. Both parameters are here, and it returns a promise with a `Result` array, even if there's just one entry.
3. We don't need any type annotations. All type annotations are defined through `SearchFn`, and TypeScript

infers the correct types: `query` becomes a string, `tags` becomes a string array. And we get red squiggly lines if we don't return a promise with values of the shape of `Result[]`.

We get the same type inference if we extract the function into its own anonymous function and assign it to a `const` that is explicitly typed with `SearchFn`:

```
const testSearch: SearchFn = function(query, tags) {
  // All types still intact
  return Promise.resolve([{
    title: `The ${query} test book`,
    url: `/${query}-design-patterns`,
    abstract: `A practical book on ${query}`
  }])
}
```

TypeScript has a structural type system. This also applies to functions: the shape has to be intact. Function shapes, however, work a little differently: structure and shape aren't defined by the names of arguments as in objects, but by the order of arguments. This means we can rename our parameters and still retain types:

```
const testSearch: SearchFn = function(term, options) {
  // term is a string (as defined by query)
  // options is an optional string array
  return Promise.resolve([{
```

```
    title: `The ${term} test book`,
    url: `/${term}-design-patterns`,
    abstract: `A practical book on ${term}`
  }])
}
```

This is especially important if different names make more sense. What if our back end doesn't work with tags but with different parts of a bigger portal, like documentation, a marketing website, or forum? A name like `sections` or `subdomains` would make more sense than the very generic name `tags`.

And we can also completely remove the optional parameter `tags` (the string array) if we don't have any use for it:

```
const testSearch: SearchFn = function(term) {
  // All types still intact
  return Promise.resolve([{
    title: `The ${term} test book`,
    url: `/${term}-design-patterns`,
    abstract: `A practical book on ${term}`
  }])
}
```

This is the power of structural type systems for functions. But you know what? We can change even more.

Lesson 17: Substitutability

In the last lesson, we saw that we can drop function parameters if the type declares this argument optional. There is a little more to that. We can also drop arguments entirely if we don't have any use for them:

```
// This is a valid search Function
const dummyContentSearchFn: SearchFn = function() {
  return Promise.resolve([{
    title: 'Form Design Patterns',
    url: '/form-design-patterns',
    abstract: 'A practical book on accessible forms'
  }])
}
```

This function satisfies the type contract by `SearchFn`, even though at first the function's shape doesn't look compatible. Except that it is. This has something to do with the way JavaScript works.

Number of Parameters

In JavaScript, we can call a function with any amount of parameters, no matter how many we define in the function's head. This can lead to two edge cases.

First, we call the function with missing arguments. Since we most likely depend on getting values inside the function's body, this causes the function to fail miserably. But this is covered by TypeScript's type checks.

If a function signature requires us to pass arguments, TypeScript will report an error if we don't – and give us red squigglies.

```
// Calling our original search function.
// TypeScript tells us that we need to pass
// at least a query
search()
```

Good! TypeScript makes sure we pass the parameters required by the function type. So calling a function is covered.

The second edge case, however, is different: we can pass *too many* parameters to a function, and the excess parameters simply get ignored. This is fine. Why should we define function parameters in our function's head if we don't have any use for them in our function's body?

JavaScript still allows us to pass the parameters; we just don't do anything with them. This makes `dummyContentSearchFn`, with no parameters, compatible with the type `SearchFn`.

A nice side effect is that since we explicitly typed `dummyContentSearchFn` to `SearchFn`, and assigned an anonymous function, we can't call `dummyContentSearchFn` without the right amount of parameters defined by `SearchFn`.

```
dummyContentSearchFn('Ember') // Good!
dummyContentSearchFn('Ember', ['JavaScript']) // Good!

// Not good, as an explicitly typed SearchFn requires
// us to pass parameters
dummyContentSearchFn()
```

If we refactor `dummyContentSearchFn` to be a named function and not explicitly typed, the behavior is fundamentally different:

```
function dummyContentSearchFn() {
  return Promise.resolve([{
    title: 'Form Design Patterns',
    url: '/form-design-patterns',
    abstract: 'A practical book on accessible forms'
  }])
}

dummyContentSearchFn('Ember') // Nope!
dummyContentSearchFn('Ember', ['JavaScript']) // Nope!

// Good!
dummyContentSearchFn()
```

This is because TypeScript again checks for the contract of the function type. And this time, the contract is to not have any arguments.

We still can pass `dummyContentSearchFn` to `displaySearch`. Inside `displaySearch`, `dummyContentSearchFn` takes on the shape of `SearchFn`, even though it does nothing with the parameters passed.

```
displaySearch('input', 'output', dummyContentSearchFn)
```

TypeScript calls this behavior *substitutability*. We can substitute one function signature for another if it makes sense. Leaving out parameters if we don't have any use for them inside the function body is OK. The code will still work. This is one of the many ways TypeScript is less strict and more pragmatic, to conform to the way JavaScript works.

void

Substitutability works because the types of the return values stay the same. In both `dummyContentSearchFn` and `testSearchFn`, we return a promise with a result array. Passed parameters disappear the moment we don't need them anymore.

There are situations where we can also substitute the return type of the function: when the return type is `void`.

`void` as a type is a curious construct in TypeScript, as it tries to mirror the behavior of `void` in other programming languages, but it has a lot more in common with `void` in JavaScript. There's a whole article on `void` in JavaScript and TypeScript on my weblog.[18]

In JavaScript, all functions have a return value. If we don't return one on our own, the return value is by default `undefined`. In TypeScript, every function has a return type. If we don't explicitly type or infer, the return type is by default `void`.

`void` in TypeScript is a different way of saying `undefined`. The `void` type can take only one value, which is `undefined`, but there are some interesting features to it.

Let's refactor, for now, the search function to not return a promise but to pass the results to a callback.

```
// We add a callback as second parameter, as
// optional parameters always have to be last
function search(
  query: string,
  callback: (results: Result[]) => void,
  tags?: string[]
) {
```

18 https://smashed.by/void

```
  let queryString = `?query=${query}`
  if(tags && tags.length) {
    queryString += `&tags=${tags.join()}`
  }
  fetch(`/search${queryString}`)
    .then(res => res.json() as Promise<Result[]>)
    // Here, we pass the results to our callback
    .then(results => callback(results))
}
```

A similar function to the original one, but the second parameter is now a callback that takes a result array and returns `void`. We can use the new search function like this:

```
// logs all results to the console
search('Ember', function(results) {
  console.log(results)
})
```

And, as we are used to with callbacks, we can pass any function that resembles the function's shape:

```
function searchHandler(results: Result[]) {
  console.log(results)
}
search('Ember', searchHandler)
```

But here's the thing: we can also pass functions that have a different return type.

```
// Search handler now returns a number
function searchHandler(results: Result[]): number {
  return results.length
}

// Totally OK!
search('Ember', searchHandler)
```

We can substitute any return type for `void`. Inside the calling function, the return type will be handled as undefined, which means you can't do anything with it that wouldn't let TypeScript scream at you with red squiggly lines:

```
function search(
  query: string,
  callback: (results: Result[]) => void,
  tags?: string[]
) {
  ...
  fetch(`/search${queryString}`)
    .then(res => res.json() as Promise<Result[]>)
    .then(results => {
      const didItWork = callback(results)
      // didItWork is undefined! This causes an error
      didItWork += 2
    })
}
```

This is also to conform to the way JavaScript works. There are occasions where we pass callback functions that return something even though the function called doesn't do anything with it, especially if you want to reuse functions over and over in different scenarios.

```
// This function shows results in an HTML element
// but also returns the container element that
// has been filled
function showResults(results: Result[]) {
  const container
    = document.getElementById('results')
  if(container) {
    container.innerHTML = `<ul>
      ${results.map(el => `<li>${el.title}</li>`)}
    <ul>`;
  }
  return container;
}

// Somewhere in our app, we show a list of
// pages on click

button.addEventListener('click', function() {
  const el = showResults(storedResults)
  if(el) {
    el.style.display = 'block'
  }
})

// But hey, this function also makes a good
// search handler
```

```
search('Ember', showResults)
```

If you really want to make sure that no value is returned, you can either put `void` in front of `callback` in plain JavaScript:

```
function search(
  query: string,
  callback: (results: Result[]) => void,
  tags?: string[]
) {
  ...

  fetch(`/search${queryString}`)
    .then(res => res.json() as Promise<Result[]>)
    // void is a keyword in JavaScript returning
    // undefined
    .then(results => void callback(results))
}
```

or use `undefined` as a type:

```
// We change the return type of callback to
// undefined
```

```
function search(
  query: string,
  callback: (results: Result[]) => undefined,
  tags?: string[]
) {
  ...
}

function searchHandler(results: Result[]): number {
  return results.length
}

// This breaks now!
search('Ember', searchHandler)
```

As we can't substitute `undefined` for `number`. Substitutability is a concept in TypeScript that you will stumble upon if you work a lot with functions. Instead of being too strict with exact function shapes, it complements the way JavaScript works with functions: asking only for the parameters that you actually need.

There are, however, some occasions when a TypeScript function declaration could have *more* arguments than its JavaScript counterpart. And it, literally, has to do with `this`.

Lesson 18: This and That

It's time to implement the `displaySearch` function, to show us another side of callbacks and functions that is unique to TypeScript. This is our function's head:

```
declare function displaySearch(
  inputId: 'string',
  outputId: 'string',
  search: SearchFn
): void;
```

We want to select elements in our markup via `document.getElementById` and

1. Get the current value out of the input field.
2. Show the results in the output element.

Let's go!

The Implementation

Let's assume that this is the necessary markup we want to enhance with our little function:

```
<form action="/search" method="POST">
```

```
  <label for="search">Search the site</label>
  <input type="search" id="search">
  <button type="submit">
</form>
<div id="output" hidden>
</div>
```

In true progressive enhancement fashion, this search field works perfectly well and leads people to a search page with all the results based on the term they entered in the search field.

But we want to enhance it with dynamic features: loading search results as we type, showing the first five in the `output` box. Giving people an idea of what to expect.

First, we select the `input` element which goes by the ID passed in the `inputId` argument. We want to listen to all `change` events that are fired.

Once a `change` event is fired, we set an active state to the entire form, which means we add a class called `active` to the parent element.

```
function displaySearch(
  inputId: string,
  outputId: string,
```

```
  search: SearchFn
): void {
  document.
    getElementById(inputId)?.
    addEventListener('change', function() {
      this.
        parentElement?.
        classList.add('active')
    })
}
```

The type annotations in the function header aside, we see regular JavaScript. And TypeScript doesn't complain about anything. No red squigglies. Everything compiles as we want it.

Does anything strike you as odd about it working so seamlessly? No? Good! This is regular, working JavaScript code as we would write it without TypeScript, so it's good that TypeScript doesn't throw red squigglies at you.

Now let's add some more code. After we set our form to `active`, we fetch the current value from the input field, to pass it to the search function.

```
function displaySearch(
  inputId: string,
  outputId: string,
```

```
  search: SearchFn
): void {
  document.
    getElementById(inputId)?.
    addEventListener('change', function() {
      this.
        parentElement?.
        classList.add('active')
      const searchTerm = this.value
    })
}
```

What's that? A red squiggly? How did that happen? Well, let's talk about this.

Function Binding and HTML Elements

In JavaScript, regular functions are always bound to an object. This object becomes accessible via `this` in these functions. In our example, the callback function is bound to the element we retrieved via `getElementById`. So `this` at this moment is the element node, which is of type `HTMLElement`.

This behavior is provided by TypeScript. The purpose of `getElementById` is to retrieve HTML elements. Each element is of a certain type – you can look them up in MDN,[19]

19 https://smashed.by/mdn

where you see a broad range for nearly every HTML element there is. From `HTMLAnchorElement` (the `a` element) to `HTMLVideoElement` (the `video` element). These are interfaces that are available in the browser.

```
// Creating an HTMLVideoElement by using
// the tag
const x = document.createElement('video')
console.log(x.toString())
// Prints "[object HTMLVideoElement]"
// the name of the actual browser interface
```

A lot of these interfaces are automatically generated by scraping web standard documents and looking for parts where interfaces are described in the web interface definition language (WIDL) format. The TSJS Generator[20] converts WIDL files to TypeScript declaration files: a wonderful window into what's going on in browser internals!

Those elements follow a very shallow inheritance hierarchy. With the supertype, the lowest common denominator is `HTMLElement`. And this is also the interface we access via `this` in our callback function.

This is also the most concrete contract the predefined TypeScript typings can provide for us. How should a static code analysis tool know which element is underneath `inputId`? When using methods like `querySelector`, the lowest com-

20 https://smashed.by/tsjs

mon denominator becomes even more generic: `Element`, which wouldn't allow us to even access `parentElement`. However, if you go for an element selector – e.g. `querySelector('input')` – TypeScript can provide a little help.

In our case, we have to check what subtype instance `this` is. This comes with another type guard check: `instanceof`.

```
function displaySearch(
  inputId: string,
  outputId: string,
  search: SearchFn
): void {
  document.
    getElementById(inputId)?.
    addEventListener('change', function() {
      // This is of type HTMLElement because
      // getElementById says so
      this.
        parentElement?.
        classList.add('active')
      if(this instanceof HTMLInputElement) {
        // From here on, this is
        // of type HTMLInputElement
        const searchTerm = this.value // Works!
        search(searchTerm)
          .then(results => {
            // TODO in another lesson
          })
      }
    })
}
```

As an additional benefit, our code becomes a lot more secure. TypeScript again points us to the things that might cause some problems: yes, we can be sure that `this` is an `HTMLElement` once we reach the callback, but we can never be completely sure that it is of type `HTMLInputElement`. What if IDs change in our markup? Our code would break into pieces and throw errors. Here, we have a guard where we can check and steer based on the outcome. Type safety in our code leads to more robust code when shipped.

Extracting the Callback

This leaves us with one problem: what if we want to extract the callback into its own function? This is not uncommon when writing JavaScript; the same function might be used at different places. But the moment we extract the function and put it in another place, we also lose any connection to `this`!

```
function inputChangeHandler() {
  // We have no clue what this can be
  // that's why we get red squigglies
  this.
    parentElement?.
    classList.add('active')
}
```

```
function displaySearch(
  inputId: string,
  outputId: string,
  search: SearchFn
): void {
  document.
    getElementById(inputId)?.
    // Only here, inputChangeHandler's this
    // becomes of type HTMLElement again
    // but inputChangeHandler doesn't know about that
    addEventListener('change', inputChangeHandler)
}
```

TypeScript has a way of dealing with situations like this: we are allowed to type `this`! Function declarations can have another additional parameter, that has to be at the very first position: `this`.

```
// We define that this is of type HTMLElement
function inputChangeHandler(this: HTMLElement) {
  this.
    parentElement?.
    classList.add('active')
}
```

This parameter gets erased once we compile TypeScript down to JavaScript. Again, there are additional benefits.

We can only use `inputChangeHandler` wherever we can be sure that this is going to be a (sub)type of `HTMLElement`. This also ensures that we don't call `inputChangeHandler` outside with no context:

```
// The 'this' context of type 'void' is not assignable
// to method's 'this' of type 'HTMLElement'.
inputChangeHandler()
```

And again, we follow the basic principle of TypeScript: being as easy to use as possible but adding enough type safety as we need to make sure we don't shoot ourselves in the foot. TypeScript puts red squigglies where we *need* to be more explicit in our thinking if we want to make sure we don't break once we run our program and subtle things change.

Lesson 19: The Function Type Tool Belt

In JavaScript you won't go far without functions. As you have seen, function *types* can be very versatile, covering a lot of use cases that come up in regular JavaScript. The mantra is always the same: JavaScript first and just enough types to give the compiler something to work with. In this lesson, we will look at a couple of function scenarios where TypeScript helps you with extra type information.

Tagged Template Literals

Among the arguably coolest features in recent JavaScript are template literals. String concatenation always was something that felt way too tedious for a modern programming language. With template literals it's simply elegant:

```
const term = 'Ember'
const results = 12

const result text =
  `You searched for ${term}, and got ${no} results`
```

We can execute any expression within a string, and concatenate the result with the rest of the string. An extension of template literals is *tagged* template literals. Syntactically they are very similar: backtick strings with JavaScript expressions, but there's a custom *tag* in front of them.

Let's think about a *highlight* tag, that allows us to replace a certain symbol within a string with some HTML elements; for example:

```
const result = {
  title: 'A guide to @@starthl@@Ember@@endhl@@.js',
  url: '/a-guide-to-ember',
  description: 'The framework @@starthl@@Ember@@
endhl@@.js
     in a nutshell'
}
```

We want to replace every `@@starthl@@` with `<mark>` tags, and `@@endhl@@` with `</mark>` closing tags. The scenario where we want to replace these strings is within the building of a results list markup: an unordered list (`<ul>`) with lots of list items (`<li>`). The list item contains the result's title, but with the highlight markers replaced. To achieve that, we create a tag called `highlight`, that works like that:

```
let markup = highlight`<li>${result.title}</li>`
```

A tag for a tagged template literal is nothing but a function that has a defined set of parameters.

1. The first parameter is a `TemplateStringsArray`, an array that contains all the strings around the expressions. In our case `<li>` and `</li>`
2. The second is a string array with the actual expressions. In our case whatever `${result.title}` gives us.

It is our task to concatenate all string parts again and, wherever we like, to make modifications; for example: replace all markers with actual elements.

TypeScript provides us with respective types for that. The type `TemplateStringsArray` is different from other string arrays as it's read-only and has a pointer to the raw array. This mirrors the actual implementation in JavaScript.

The implementation might look something like this:

```
function highlight(
  strings: TemplateStringsArray,
  ...values: string[]
) {
  let str = '' // The result string
  strings.forEach((templ, i) => {
    // Fetch the expression from the same position
    // or assign an empty string
    let expr = values[i]?
      .replace('@@start@@', '<em>')
      .replace('@@end@@', '</em>') ?? ''
    // Merge template and expression
    str += templ + expr
  });
  return str
}
```

With a function head like this, TypeScript recognizes `highlight` as a template tag. In use, a `highlight` tagged template literal looks something like this:

```
function createResultTemplate(results: Result[]):
string {
  return `<ul>
    ${results.map(result =>
      highlight`<li>${result.title}</li>`)}
  </ul>`
}
```

Handy and, with the use of a specific function head, very intentional. That's what we want!

Rest Parameters

You just saw a very weird notation for the second string array in `highlight`. What about those dots?

```
declare function highlight(
  strings: TemplateStringsArray,
  ...values: string[]
): string
```

The dots tell us that `values` is a rest parameter. As we established in other lessons, functions in JavaScript can be called with a virtually unlimited number of parameters. The function implementation takes care of the number of parameters that are actually used.

But what if we want to use *all* parameters, no matter how many? This is where rest parameters come into play. Let's look at a different use of our search function:

```
// I want to search for a term
search('Ember')
```

```
// I want to add a tag
search('Ember', 'JavaScript')

// and a second tag
search('Ember', 'JavaScript', 'Web Development')

// and so on...
search('Ember', 'JavaScript', 'Web Development',
'Code')

// and so on...
search('Ember', 'JavaScript', 'Web Development',
  'Code', 'Guides')
```

We as developers see one search term, and as many tags as we like. How do we declare types for that in TypeScript? With arrays!

```
declare function
  search(term: string, ...tags: string[]):
Promise<Result[]>
```

Just by adding three dots, the usage of the search function varies, while the way the search function works internally stays exactly the same. Note that rest parameters are always

optional. So even if `term` and `tags` have the same type, you want to carve out as many parameters as you need to make your function work.

Asynchronous Functions

Functions that return promises are functions we can use in an asynchronous context, which means we can `await` their result:

```
const results = await search('Ember') // Yass!
```

Which means that we can declare functions asynchronous. Using the `async` keyword affects the function body and the implementation. Your return values are automatically wrapped in a promise return type:

```
async function search(
  query: string,
  tags?: string[]
) {
  let queryString = `?query=${query}`
  if(tags && tags.length) {
    queryString += `&tags=${tags.join()}`
  }

  // Instead of thenable promise calls
  // we await results
```

```
  const response
    = await fetch(`/search${queryString}`)
  const results = await response.json()

  // The return type becomes Promise<Result[]>
  return results as Result[]
}
```

Note that TypeScript supports top-level `async`. This means that as long as you are in a module context, you can call `async` functions as much as you like.

One important detail: when you `declare` an asynchronous function type, you can't use the `async` keyword. The reason is that the function head stays exactly the same. We return a promise of results – nothing has changed compared with the previous version.

```
declare function
  search(term: string, tags?: string[]):
Promise<Result[]>
```

Once you declare the function type like that, you can use this function in an asynchronous context.

Lesson 20: Function Overloading

In the previous lesson we saw a couple of different implementations of the `search` function. Let's review. First, the traditional. It takes a term and some optional tags, and returns a promise with results:

```
declare function search(
  term: string,
  tags?: string[]
): Promise<Result[]>
```

Second, we had a search function that takes a term, a callback, and optional tags as a third parameter. We don't return values, only `void`.

```
declare function search(
  term: string,
  callback: (result: Result[]) => void,
  tags?: string[]
): void
```

There might be more variations, but let's focus on those two for now. Those two function heads look entirely different,

except for the first parameter. They return different things, and they take arguments in a different order, or with different types entirely.

However, it isn't that uncommon in JavaScript to have functions that take differently typed arguments at different positions. Take a look at the `write` function of the file system API in Node.js.[21]

The second argument can be either a buffer with data or a string that we write to a file. And there are lots of optional parameters afterwards. The only thing that's constant is that the last argument is always a callback. But this callback can be at position three, four, five, or even six!

In JavaScript, function arguments are what we make of them. This is fundamentally different from some other programming languages that allow for multiple functions within the same scope. Other programming languages, like C++, Java, or C#, call this *function overloading*: having more than one implementation as long as the argument list is different.

In JavaScript, we can only have one function with a specific name in a specific scope. But the argument list can be as dynamic as we need it. So functions in JavaScript can do basically anything and everything. But how do we type this?

21 https://smashed.by/fswrite

This is where TypeScript borrows the term *function overloading* from other programming languages: if your function can have anything as arguments, you can at least write type definitions for different use cases. Instead of having multiple implementations, you have one implementation, but more types.

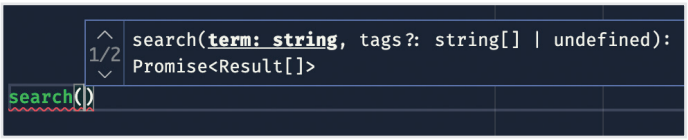
In TypeScript, we define types like that by writing the function overloads on top of the actual implementation. For our search function, we already know which types we need; we declared them at the beginning of this lesson:

```
function search(
  term: string,
  tags?: string[]
): Promise<Result[]>
function search(
  term: string
  callback: (results: Result[]) => void,
  tags?: string[]
): void
```

After the two function declarations, we need the *actual* function that implements our search function. The function head has to be very special: the argument list has to encompass all argument lists from all function overloads above. Since most of the arguments are different, we use `any` or `unknown` to type them.

```
function search(
  term: string,
  tags?: string[]
): Promise<Result[]>
function search(
  term: string,
  callback: (results: Result[]) => void,
  tags?: string[]
): void
// Here comes the implementation
function search(
  term: string,
  p2?: unknown,
  p3?: string[]
) {
  // Now for the implementation
}
```

The moment we define our search function like that, we get autocompletion for the two function overloads.



Visual Studio code showing the two possible function heads for the search function.

How did we come up with the final function head?

1. The first argument, `term`, is the same in both function heads. We carry over that argument.
2. The second argument, `p2`, can be optional `tags`, or a callback. So it is definitely optional, and it can have two types. That's why we choose `unknown` for now.
3. The third argument, `p3`, *must* be optional as the second argument can be optional. We can't do required parameters after optional parameters, obviously. But once argument three appears, it's definitely `tags` from the second function overload. Hence, `string[]`.

We deliberately choose undescriptive names like `p2` or `p3` to redefine them to their actual purpose inside the function body. We can also name `p2` something like `tagsOrCallback`, and `p3` to `tagsAfterCallback` if you want to be more intentional.

It's good practice to get intentional about your overloaded arguments at the very beginning of your function body:

```
function search(
  term: string,
  tags?: string[]
): Promise<Result[]>
function search(
  term: string,
  callback: (results: Result[]) => void,
  tags?: string[]
```

```
): void
function search(
  term: string,
  p2?: unknown,
  p3?: string[]
) {
  // We only have a callback if 'p2' is a function
  const callback =
    typeof p2 === 'function' ? p2 : undefined

  // We have tags if p2 is defined and an array, or if p3
  // is defined and an array
  const tags =
    typeof p2 !== 'undefined' && Array.isArray(p2) ? p2 :
    typeof p3 !== 'undefined' && Array.isArray(p3) ? p3 :
    undefined;

  let queryString = `?query=${term}`

  if(tags && tags.length) {
    // tags at this point has to be an array
    queryString += `&tags=${tags.join()}`
  }

  // The actual fetching of results!
  const results = fetch(`/search${queryString}`)
    .then(response => response.json())

  // callback is either undefined or a function, as
  // seen above
  if(callback) {
    // Now it's definitely a function! So let's then()
    // the results and call the callback!
```

```
    // We don't return anything. This is equivalent to
    // void
    return void results.then(res => callback(res))
  } else {
    // Otherwise, we have to return a promise with
    // results as described in the first function
    // overload
    return results
  }
}
```

There's a lot to do if you have to take care of implementing two different function heads at once. It gets even more complicated the more overloads you have. If possible, we shouldn't overload too much. As Shawn Wang[22] once put it, if it's difficult to implement in TypeScript, it may be difficult to use without TypeScript.

Still, TypeScript will recognize the search function to be compatible with everything that needs the promise-based version, and everything that needs the callback variant.

Note that inside the function body we lose a lot of type information due to `p2` being `unknown`. We can narrow it down to a regular function, or to an array of `any`. If we want to know more about the types inside the function body, and still satisfy the function overloads, we can be more explicit:

22 https://smashed.by/swyx

```
function search(
  term: string,
  tags?: string[]
): Promise<Result[]>
function search(
  term: string,
  callback: (results: Result[]) => void,
  tags?: string[]
): void
function search(
  term: string,
  p2?: string[] | ((results: Result[]) => void,
  p3?: string[]
) {
  // All from above, but with better type info
}
```

This construct is called a *union type*, where we define this parameter as either a string array or a function that accepts an array of results as a parameter. We are going to talk a lot about union types in chapter 4.

Function Types with Overloads

In the previous lessons we found out that we can create function type aliases to declare the contract in a single type. Like `SearchFn` earlier on. We can do the same thing with

overloaded functions. A quick way is to get the type via `typeof search` again, and it might look something like this:

```
type SearchOverloadFn = {
  // Function overload number 1
  (
    term: string,
    tags?: string[] | undefined
  ) : Promise<Result[]>;
  // Function overload number 2
  (
    term: string,
    callback: (results: Result[]) => void,
    tags?: string[] | undefined
  ): void;
}
```

We can use this function type again to type arrow functions that react to overloads:

```
const searchWithOverloads: SearchOverloadFn =
  (
    term: string,
    p2?: string[] | (results: Result[]) => void,
    p3?: string[]
  ) => {
    // Do your magic
  }
```

Note that type inference is hard for multiple variations. That's why you have to provide the types for the parameters in the arrow function yourself.

Lesson 21: Generator Functions

In JavaScript, there is a special kind of function called a generator function. Generator functions can be exited and later reentered. The idea is that such a function *generates* values over the course of time, hence its name.

That sounds incredibly complicated and, truth be told, they are. But TypeScript's typing information can help you a lot in developing them. And the type inference is pretty great!

There are two things to remember when working with generators.

1. There's an asterisk around the generator function telling you that this is not an ordinary function.
2. There's a new keyword: `yield`. It acts as a doorway that passes results to the outside, but also allows us to enter values for the next iteration. If you ever saw

a function as a black box, see `yield` as a hatch where we can peek inside.

A very simple generator function can look like this:

```
// Nonsensical, but it illustrates the way they work
function *generateStuff() {
  yield 1
  yield 2
  let proceed = yield 3
  if(proceed) {
    yield 4
  }
  return 'done'
}

// In use:
const generator = generateStuff()
console.log(generator.next().value) // logs 1
console.log(generator.next().value) // logs 2
console.log(generator.next().value) // logs 3
// The door is open, we pass true through and...
console.log(generator.next(true).value) // logs 4
console.log(generator.next().value) // 'done'
```

`yield` first opens a door and returns a value. The open door then accepts a value, passed in the next function. That's why we were able to pass `true` at step four, because the door was open when we yielded 3, but before yielding 4.

The typings for that are already very complicated, but TypeScript can already infer a lot. If you hover over the function, you see that TypeScript infers the following return type:

```
Generator<1 | 2 | 3 | 4, string, unknown>
```

which means that we yield the numbers 1, 2, 3, or 4. We will return a string eventually, and we don't have any idea what comes inside once we open the door. Truth be told, this generator function is probably not the most useful one, and the types mirror that. In a more realistic example, we will see that types for generator functions can tell us a lot if we do them right.

Polling Search

In our example we want to have a polling search. Imagine a back end that reacts to a specific search query and then returns results within milliseconds. The results are by no means complete, just a few it was able to fetch from the database. It also tells us if it's finished with the query. We then have the possibility to query again and get more results. We constantly *poll* the back end for more.

This already sounds like a type:

```
type PollingResults = {
  results: Result[],
  done: boolean
};
```

A very simplified implementation of the polling function can look like this. We strip away things like resetting the search query or connecting results to a user:

```
async function polling(
  term: string
): Promise<PollingResults> {
  return fetch(`/pollingSearch?query=${term}`)
    .then(res => res.json())
}
```

The reason why we're fetching results in batches is that we want to show them as soon as possible, appending results on the go:

```
function append(result: Result) {
  const node = document.createElement('li')
  node.innerHTML = `
    <a href="${result.url}">${result.title}</a>
  `
  document.querySelector('#results')?.append(node)
}
```

This is where a generator function comes in handy. We constantly poll our back end and return results, but end the function only if the back end tells us it's done.

```
async function *getResults(term: string) {
  let state
  do {
    // state is a PollingResult
    state = await polling(term)
    // yield the current result array
    yield state.results
  } while(!state.done)
  // Nothing more to do
}
```

And that's our very simple, very basic generator. Note that for now, we only had to set the type of the input value. So far, all the other things have been inferred. We also leveled up the game: we're working now with async generators. This shouldn't concern us too much, the `async` keyword hides most complexity.

The type of the generator is as follows:

```
AsyncGenerator<Result[], void, unknown>
```

1. We yield `Result[]` arrays.

2. We return nothing, hence `void`.
3. We don't pass anything. Unknown values come through the door.

Now for the usage of our generator. Each generator returns an *iterator*, a way to loop through values. Each call to `next` returns an iterator result with a possible `value`, and a status (if it's `done`). So looping through our constantly fetched results looks something like this:

```
// Adding an event listener, we've been there
document.getElementById('searchField')?.
  addEventListener('change', handleChange)

// The actual event handler
async function handleChange(this: HTMLElement, ev:
Event) {
  if (this instanceof HTMLInputElement) {
    // Search for a term,
    // call the generator, get an iterator
    let resultsGen = getResults(this.value);
    let next
    do {
      // Get the next iterator result
      next = await resultsGen.next()

      // The value can be a Result[] or void
      // because that's what the generator function
      // returns
      if(typeof next.value !== 'undefined') {
        next.value.map(append)
```

```
      }
    } while(!next.done) // As long as we are not done
  }
}
```

For this very basic iteration, where we don't put anything back through the `yield` door, we can use `for await` loops:

```
async function handleChange(this: HTMLElement, ev:
Event) {
  if (this instanceof HTMLInputElement) {
    let resultsGen = getResults(this.value);
    for await(results of resultsGen) {
      results.map(append)
    }
  }
}
```

Much clearer! But guess what? We *want* to put something back through the `yield` door.

Yielding In

The great thing about having a generator function is that we can control the output midway through. Of course, we

can wait until our back end is complete with all the search results and sends us the `done` flag. Or we preemptively say stop polling if we reach a certain amount of results we want to show.

Thankfully, the `next` function allows us to pass results in. A call if the polling should proceed would be nice, preferably when we show more than five results. The `handleChange` function is adapted quickly by introducing a counter variable.

```
async function handleChange(this: HTMLElement, ev:
Event) {
  if (this instanceof HTMLInputElement) {
    let resultsGen = getResults(this.value)
    let next
+   let count = 0
    do {
-     next = await resultsGen.next()
+     next = await resultsGen.next(count >= 5)
      if(typeof next.value !== 'undefined') {
        next.value.map(append)
+       count += next.value.length
      }
    } while(!next.done)
  }
}
```

TypeScript is OK with this change as everything we can pass in through the `yield` door is `unknown`. So we have to make that more concrete. Let's get back into our generator function.

```
async function *getResults(term: string) {
  let state
+ let stop
  do {
    state = await polling(term)
-   yield state.results
+   stop = yield state.results
- } while(!state.done)
+ } while(!state.done || stop)
}
```

This works, but now the worst case has happened: we went from `unknown` (good) to `any` (very bad). Let's be more concrete. Either let TypeScript infer it by assigning a default value:

```
async function *getResults(term: string) {
  let state
+ let stop = false
  do {
    state = await polling(term)
    stop = yield state.results
  } while(!state.done || stop)
}
```

This already changes the type to

```
AsyncGenerator<Result[], void, boolean>
```

And this makes sure that we pass the correct types. Or, we are very explicit about our return type:

```
async function *getResults(
  term: string
): AsyncGenerator<Result[], void, boolean> {
  let state, stop
  do {
    state = await polling(term)
    stop = yield state.results
    // from here on, stop is boolean
  } while(state.done && stop)
}
```

Like so often, the choice is between casually going forward as we code, or being very explicit about contracts by defining function heads as concretely as possible.

Recap

Functions in JavaScript are big. So too in TypeScript. In this chapter, we've learned a lot about functions:

1. First, we learned about function types, return types, and parameter types.
2. We dug into callbacks, a concept in JavaScript that pops up everywhere. We learned that functions have

their own types, and that argument order is important, rather than their names.

3. We learned about the concept of substitutability. Functions can have a different shape than their types if the context allows for it.
4. Sometimes when writing JavaScript, I want to shout, "This is ridiculous!" but I never know what `this` refers to. Well, thank goodness TypeScript can help us with that! `this` argument types help us to prevent errors and get more info about the object we bind our function to.
5. We learned how TypeScript infers async function return types and works with rest parameters.
6. Also, we saw that TypeScript needs special function heads for tagged template literals.
7. Function overloads help us to define multiple function types for one function, comforming to the flexibility of JavaScript functions, but also making the complexity of very flexible functions visible.
8. Last, but not least, we dug into a special kind of function: the ones with the asterisk, generator functions!

We also took a peek into some more advanced concepts like union types, generics, and much more! Stuff to unravel in the chapters to come.

Interlude: Anders Hejlsberg

To fully understand the purpose of TypeScript, it helps to understand the work of TypeScript's lead architect, **Anders Hejlsberg**.

Anders Hejlsberg is a programming language designer who has had a huge influence on programming languages and their tools over the last four decades. Some people say Anders redefined programming with every major programming language he created.

Hejlsberg started out developing his version of the Pascal programming language back in the early 1980s. He called it Turbo Pascal, and the name alone indicates the key difference to existing dialects. Pascal had been implemented by a lot of companies at that time, but what made Anders' implementation so different from the rest was the focus on developer experience.

Back in the 1980s, compiling took quite some time. Two compiler passes were necessary to transform source code into binary code. A subsequent linker pass assembled an executable that was runnable on the selected system. It could take anything from several minutes to a few hours before developers were able to see results. Sometimes floppy disks

had to be switched over and over again. If there was an error, the whole process had to start again.

Enter Turbo Pascal. Hejlsberg took the Pascal programming language and stripped it down to its most essential features. He created a compiler that was highly optimized and incredibly fast. With that, the usual compile-link-debug cycle was reduced to mere seconds.

This allowed for something really important: the inclusion of an integrated development environment (IDE), which could compile on save, show errors at the source-code level without exiting the program, and provide excellent feedback during development cycles to improve productivity.

This focus on excellent tooling, combined with a very reasonable and inclusive licence made Turbo Pascal a smash hit among developers. Turbo Pascal became very popular and it's still taught at universities as an entry-level programming language. It was the first programming language I learned when taking a programming course in high school.

In the early 1990s, Hejlsberg took the philosophy of great developer experience and high-quality tooling to develop

Delphi, an object-oriented version of Pascal that came with WYSIWYG editors for Windows 3.1 UIs. If you have ever used the original WinAPI for C, you know how revolutionary a drag-and-drop editor for UIs was at that time.

Later he developed C#. Criticized by many as a mere Java clone, C# took Java's shortcomings and made them more understandable on a language level and more usable on an editing level.

You can see a pattern here. Anders Hejlsberg never created the most innovative programming languages that introduced revolutionary syntax or fundamentally new programming concepts. Anders took good things from popular languages and made them fun to use.

Hejlsberg did the same thing to JavaScript with TypeScript. He and his team looked very closely at what JavaScript had to offer, and also at ideas that were on the horizon for ECMAScript (the standard behind JavaScript) and future implementations of the language.

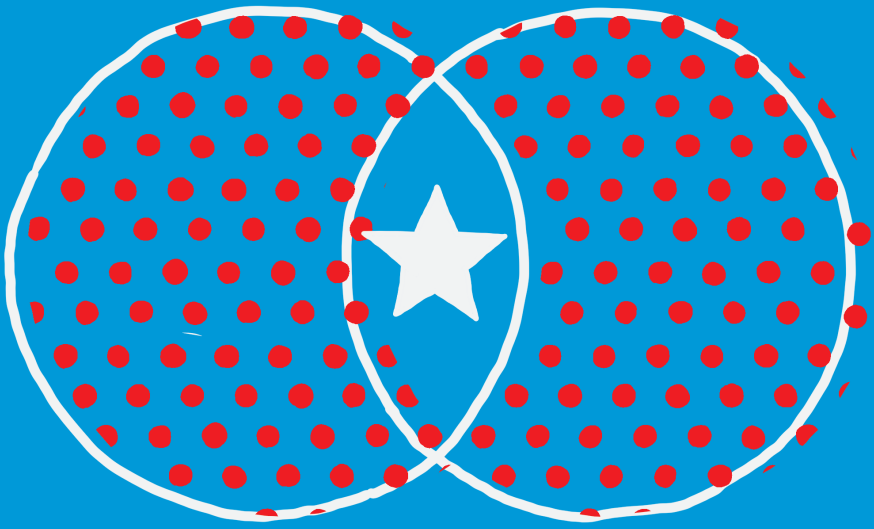
The idea of adding classes, modern syntax, and type definitions to JavaScript was not an entirely new concept. Lots of ideas have already been designed in ECMAScript 4,[23] the abandoned standard that wanted too much.

23 https://smashed.by/ecmascript

New language features to TypeScript were – and are, most of the time – something the ECMAScript community is already discussing for JavaScript. The TypeScript team is very much involved in language discussions for JavaScript, and it tries to provide a safe testing ground for new features before they hit browsers.

The type system itself, the one feature that gave TypeScript its name and helps us get excellent tooling in editors, was heavily inspired by a proposed type system in ECMAScript 4, as well as early implementations in ECMAScript dialects, such as Adobe's ActionScript (Flash veterans remember).

TypeScript is guided by the same core principles adopted by Anders Hejlsberg for his other projects: innovate in the programming language where necessary, and provide excellent tooling and editor experience alongside.



Chapter Four

UNION AND INTERSECTION TYPES

Union and Intersection Types

We've come quite far with TypeScript. We've learned about the tooling aspect, type inference, and control flow analysis, and we know how to type objects and functions effectively. With what we have learned, we are able to write pretty complex applications and most likely will get good enough tooling out of it to get us through our day.

But JavaScript *is* special. The flexibility of JavaScript that allows for easy to use programming interfaces is, frankly, hard to sum up in regular types. This is why TypeScript offers a lot more.

Starting with this chapter, we'll go deep into TypeScript's type system. We will learn about the set theory behind TypeScript, and how thinking in *unions* and *intersections* will help us get even more comprehensible and clearer type support. This is where TypeScript's type system really shines and starts becoming much more powerful than what we know from traditional programming languages. It's going to be an exciting ride!

To illustrate the concepts of union and intersection types, we'll work on a page for tech events: meetups, conferences, and webinars; events that are similar in nature, but distinct enough to be treated differently.

Lesson 22: Modeling Data

Imagine a website that lists different tech events:

1. **Tech conferences:** people meet at a certain *location* and listen to a couple of *talks*. Conferences usually cost something, so they have a *price*.
2. **Meetups:** smaller in scale, meetups are similar to conferences from a data perspective. They also happen at a certain *location* with a range of *talks*, but compared with tech conferences they are usually *free*. Well, at least in our example they are.
3. **Webinars:** instead of people attending in a physical space, webinars are online. They don't need a location, but a URL where people can watch the webinar in their browser. They can have a price, but can also be free. Compared with the other two event types, webinars feature only one *talk*.

All tech events have common properties, like a date, a description, a maximum number of attendees, and an RSVP count. We also have a string identifier in the property *kind*, where we can distinguish between conferences, webinars, and meetups.

In our app, we're working with that kind of data *a lot*. We grab a list of tech events as JSON from a back end, and also

when we add new events to a list, or want to retrieve their properties to display them in a UI.

To make life easier – and much less prone to errors – we want to spend some time modeling this data as TypeScript types. With that, we not only get proper tooling but also red squiggly lines should we forget something.

Let's start with the easy part. Every kind of tech event has some sort of talk, maybe several. A talk has a title, an abstract, and a speaker. We keep the speaker simple for now and represent them with a simple string. The type for a talk looks like this:

```
type Talk = {
  title: string,
  abstract: string,
  speaker: string
}
```

With that in place, we can develop a type for conferences:

```
type Conference = {
  title: string,
  description: string
  date: Date,
  capacity: number,
  rsvp: number,
  kind: string,
```

```
  location: string,
  price: number,
  talks: Talk[]
}
```

... a type for meetups, where price is a string ("free") instead of a number:

```
type Meetup = {
  title: string,
  description: string
  date: Date,
  capacity: number,
  rsvp: number,
  kind: string,
  location: string,
  price: string,
  talks: Talk[]
}
```

... and a type for webinars, where we only have one talk, and we don't have a physical location but a URL to host the event:

```
type Webinar = {
  title: string,
  description: string
  date: Date,
```

```
  capacity: number,
  rsvp: number,
  kind: string,
  url: string,
  price?: number,
  talks: Talk
}
```

Also, you see that types are optional. With those four types in place, we already modeled a good part of the possible data we can get from the back end. And some parts have a common shape within all three event types, and other parts are subtly, or entirely, different.

Intersection Types

The first thing we realize is that there are lots of similar properties; properties that also should stay the same, the basic shape of a `TechEvent`. With TypeScript, we're able to extract that shape and combine it with properties specific to our concrete single types.

First, let's create a `TechEventBase` type that contains all the properties that are the same in all three event types.

```
type TechEventBase = {
```

```
  title: string,
  description: string
  date: Date,
  capacity: number,
  rsvp: number,
  kind: string
}
```

Then, let's refactor the original three types to combine `TechEventBase` with the specific properties of each type.

```
type Conference = TechEventBase & {
  location: string,
  price: number,
  talks: Talk[]
}

type Meetup = TechEventBase & {
  location: string,
  price: string,
  talks: Talk[]
}

type Webinar = TechEventBase & {
  url: string,
  price?: number,
  talks: Talk
}
```

We call this concept *intersection types*. We read the `&` operator as *and*. We combine the properties from one type A with that of another type B, much like extending classes. The result is a new type with the properties of type A *and* type B.

The immediate benefit we get is that we can model common properties in one place, which makes updates and changes a lot easier.

Furthermore, the actual difference between types becomes a lot clearer and easier to read. Each subtype has just a couple of properties we need to take care of, instead of the full list.

Union Types

But what happens if we get a list of tech events, where each entry can be either a webinar, or a conference, or a meetup? Where we don't know exactly what entries we get, only that they are of one of the three event types.

For situations like that, we can use a concept called *union types*. With union types we can model exactly the following scenario: defining a `TechEvent` type that can be either a `Webinar`, or a `Conference`, or a `Meetup`. Or, in code:

```
type TechEvent = Webinar | Conference | Meetup;
```

We read the pipe operator `|` as *or*. What we get is a new type, a type that tries to encompass all possible properties available from the types we set in union.

The new type can access the following properties:

- `title`, `description`, `date`, `capacity`, `rsvp`, `kind` – the properties all three types have in common with their original primitive type. This is what the shape of `TechEventBase` gives us.
- `talks`. This property can be either a single `Talk`, or an array `Talk[]`. Its new type is `Talk | Talk[]`.
- `price`. The `price` property is also available in all three original object types, but its own type is different. `price` can be either `string` or `number`, and – following `Webinar` – it can be optional. To safely work with price, we have to do some checks within our code: we have to check if it's available, and then we have to do `typeof` checks to see if we're dealing with a `number` or a `string`.

Working with `price` and `talks` might look something like this:

```
function printEvent(event: TechEvent) {
  if(event.price) {
    // Price exists!
    if(typeof event.price === 'number') {
      // We know that price is a number
      console.log('Price in EUR: ', event.price)
    } else {
      // We know that price is a string, so the
      // event is free!
      console.log('It is free!')
    }
  }
  if(Array.isArray(event.talks)) {
    // talks is an array
    event.talks.forEach(talk => {
      console.log(talk.title)
    })
  } else {
    // It's just a single talk
    console.log(event.talks.title)
  }
}
```

Does this structure remind you of something? Back in chapter 2 we learned about the concept of control flow, and narrowing down types with type guards. This is exactly what's happening here. Since the type can take on different shapes, we can use type guards (`if` statements) to narrow down the *union* type to its single type.

Please note that we are moving between the *union* types of the respective properties `price` and `talks`. All other information of the original types `Webinar`, `Conference`, and `Meetup` that can't be unified (like location and URL) are dropped from the shape of the union. We need some more information to narrow down to the original object shapes.

Lesson 23: Moving in the Type Space

Before we continue, let's quickly review what we've just learned. We learned about intersection types, the way to combine two or more types into one, much like extending from an object type.

And we learned about union types, a way to extract the lowest common denominator of a set of types. But why do we call them *intersection* and *union* types?

Set Theory

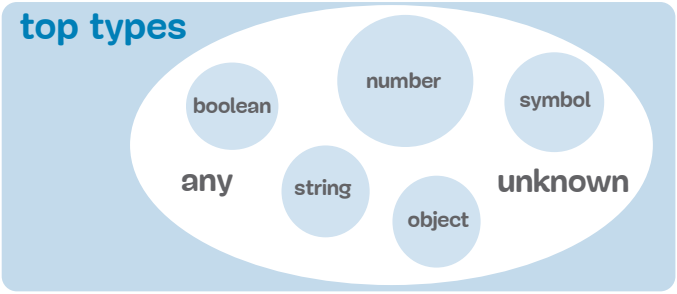
To find out, we need to review what types actually are. In his book *Programming with Types*, Vlad Riscutia defines a type as follows:[24]

24 https://smashed.by/typingintro

A type is a classification of data that defines the operations that can be done on that data, the meaning of the data, and the set of allowed values.

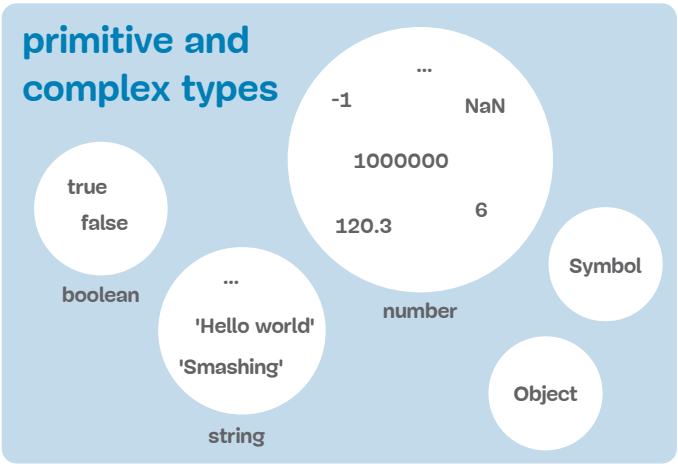
The part we want to focus on is the "set of allowed values." This is something we already experienced when working with types. Once a variable has a certain type annotation, TypeScript only allows a specific set of values to be assigned.

Type `string` only allows for strings to be assigned; `number` only allows for numbers to be assigned. Each type deals with a distinct set of values. When we think further, we can put those sets in a hierarchy. The types `any` and `unknown` encompass the whole set of all available values. They are known as **top types** as they are on the very top of the hierarchy.



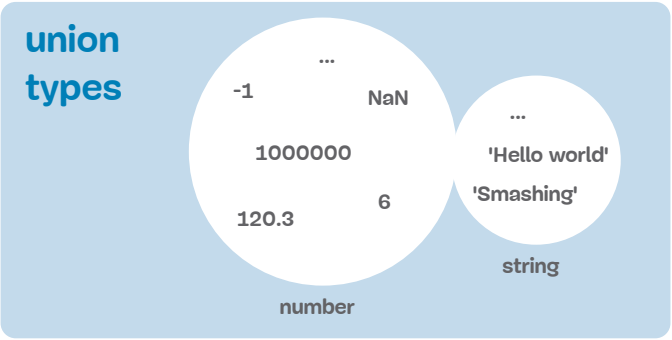
Top types, including all other types.

Primitive types such as `boolean`, `number` or `string` are one level below `any` and `unknown`. They cluster the set of all available values into distinct sets of specific values: all Boolean values, all numbers, all strings.



Primitive and complex type sets.

Those sets are distinct. They don't share any common values. If we now build a **union type** `string | number`, we allow for all values that are either from the set `string` or the set `number`, which means we get a union set of possible values.



A union of numbers and string.

If we were to build an **intersection type** `string & number`, we'd have an empty intersection set as they don't share any common values.

This is also where the term *narrowing down* comes from. We want to have a narrower set of values. If our type is `any`, we can do a `typeof` check to narrow down to a specific set in the type space. We move from a top type down to a narrower set of values.

Object Sets

With primitive types it's straightforward, but it gets a lot more fun if we consider **object types**. Take these two types, for example:

```
type Name = {
  name: string
}
type Age = {
  age: number
}
```

Since we have a structural type system, an object like

```
const person = {
  name: 'Stefan Baumgartner',
  city: 'Linz'
}
```

is a valid value of type `Person`. This object

```
// In my midlife crisis, I don't use semicolons
// ... just like the cool kids
const midlifeCrisis = {
  age: 38,
  usesSemicolons: false
}
```

is a valid value of type `Age`. This object

```
const me = {
  name: 'Stefan Baumgartner',
```

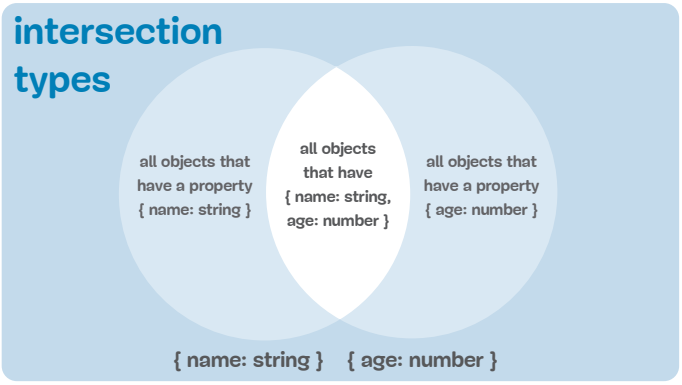
```
  age: 38
}
```

is compatible with both `Age` and `Name`.

However, we can't assign every value of type `Age` to a type `Name` because the sets are distinct enough to not have any common values. Once we define the union type `Age | Name`, both `midlifeCrisis` and `person` are compatible with the newly created type.

The set gets wider, the number of compatible values gets bigger. But we also lose clarity.

Conversely, an intersection `type Person = Age & Name` combines both sets. Now we need all properties from type `Age` and type `Name`.



An intersection of Name and Age.

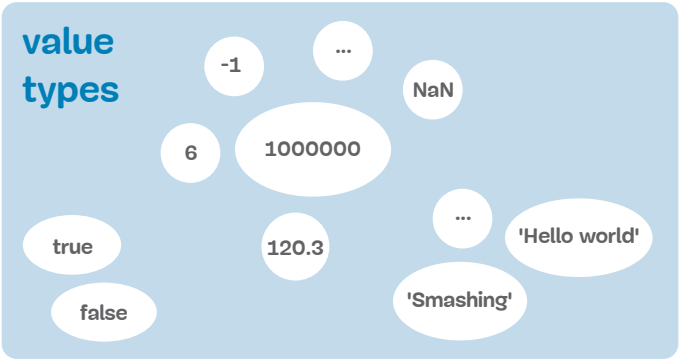
With that, only the variable `me` becomes compatible with the newly generated type. The intersection is a subset of both `Age` and `Name` sets – smaller, narrower, and we have to be more explicit about our values.

> Formally speaking, all values from type `A` are compatible with type `A | B`, and all values from type `A & B` are compatible with type `B`.

Value Types

Let's take this concept of narrowing and widening sets even further. We now know that we can have all available values and narrow them down to their primitive types. We can narrow down the complex types, like the set of all available objects, to smaller sets of possible objects defined on their property keys. Can we get even smaller?

We can! We can narrow down primitive types to values. It turns out that each specific value of a set is its own type: a **value type**.



And finally, value types.

Let's look at the string `'conference'` for example.

```
let conf = 'conference'
```

Our variable `conf` is compatible with a couple of types:

```
let withTypeAny: any = 'conference' // OK!
let withTypeString: string = 'conference' // OK!

// But also:

let withValueType: 'conference' = 'conference'
// OK!
```

You see that the set gets narrower and narrower. Type `any` selects all possible values, type `string` all possible strings. But type `'conference'` selects the specific string `'conference'`. No other strings are compatible.

TypeScript is aware of value types when assigning primitive values:

```
// Type is string, because the value can change
let conference = 'conference'

// Type is 'conference', because the value can't
// change anymore.
const conf = 'conference'
```

Now that we've narrowed down the set to value types, we can create wider custom sets again.

Let's get back to our tech events example. We have three different types of tech event: conferences, webinars, and meetups.

When our back end sends along details of which kind of events we are dealing with, we can create a custom union type:

```
type EventKind =
  'webinar' | 'conference' | 'meetup'
```

With that, we can be sure we don't assign any values that aren't intended, and we rule out typos, and other mistakes.

```
// Cool, but not possible
let tomorrowsEvent: EventKind = 'concert'
```

The value sets of primitive types are technically infinite. We would never be reasonably able to express the full spectrum of `string` or `number` in a custom type. But we can take very specific slices out of it when it conforms to our data.

> When we are deep in TypeScript's type system, we do a lot of set widening and narrowing. Moving around in sets of possible values is key to define clear yet flexible types that give us first-class tooling.

Lesson 24: Working with Value Types

Let's incorporate our new knowledge about value and union types to our tech event data structure. In lesson 22 (at the

start of this chapter) we figured out a `TechEventBase` type that includes all common properties of each tech event:

```
type TechEventBase = {
  title: string,
  description: string
  date: Date,
  capacity: number,
  rsvp: number,
  kind: string
}
```

The last property of this type is called `kind` and it holds information on the kind of tech event we are dealing with. The type of `kind` is `string` at the moment, but we know that this type can only take three distinct values:

```
type TechEventBase = {
  title: string,
  description: string
  date: Date,
  capacity: number,
  rsvp: number,
  kind: 'conference' | 'meetup' | 'webinar'
}
```

That's already much better than the previous version. We are more secure against wrong values and typos. This has an immediate effect on what we can do with the combined union type `TechEvent`. Let's look at another function called `getEventTeaser`:

```
function getEventTeaser(event: TechEvent) {
  switch(event.kind) {
    case 'conference':
      return `${event.title} (Conference)`
    case 'meetup':
      return `${event.title} (Meetup)`
    case 'webinar':
      return `${event.title} (Webinar)`
    // Again: cool, but not possible
    case 'concert':
  }
}
```

TypeScript immediately reports an error, because the type `'concert'` is not comparable to type `'conference' | 'meetup' | 'webinar'`. Unions of value types are brilliant for control flow analysis. We don't run into situations that can't happen, because our types don't support such situations. All possible values of the set are taken care of.

Discriminated Union Types

But we can do more. Instead of putting a union of three value types at `TechEventBase`, we can move very distinct value types down to the three specific tech event types. First, we drop `kind` from `TechEventBase`:

```
type TechEventBase = {
  title: string,
  description: string
  date: Date,
  capacity: number,
  rsvp: number,
}
```

Then we add distinct value types to each specific tech event.

```
type Conference = TechEventBase & {
  location: string,
  price: number,
  talks: Talk[],
  kind: 'conference'
}

type Meetup = TechEventBase & {
  location: string,
  price: string,
  talks: Talk[],
  kind: 'meetup'
```

```
}

type Webinar = TechEventBase & {
  url: string,
  price?: number,
  talks: Talk,
  kind: 'webinar'
}
```

At first glance, everything stays the same. If you hover over the `event.kind` property in our switch statement, you'll see that the type for `kind` is still `"conference" | "meetup" | "webinar"`. Since all three tech event types are combined in one union type, TypeScript creates a proper union type for this property, just as we would expect.

But underneath, something wonderful happens. Where before TypeScript just knew that some properties of the big `TechEvent` union type existed or didn't exist, with a specific value type for a property we can directly point to the surrounding object type.

Let's see what this means for the `getEventTeaser` function:

```
function getEventTeaser(event: TechEvent) {
  switch(event.kind) {
    case 'conference':
```

```
      // We now know that I'm in type Conference
      return `${event.title} (Conference), ` +
      // Suddenly I don't have to check for price as
      // TypeScript knows it will be there
        `priced at ${event.price} USD`
    case 'meetup':
      // We now know that we're in type Meetup
      return `${event.title} (Meetup), ` +
      // Suddenly we can say for sure that this
      // event will have a location, because the
      // type tells us
        `hosted at ${event.location}`
    case 'webinar':
      // We now know that we're in type Webinar
      return '${event.title} (Webinar), ' +
      // Suddenly we can say for sure that there will
      // be a URL
        `available online at ${event.url}`
    default:
      throw new Error('Not sure what to do with
that!')
  }
}
```

Using value types for properties works like a hook for TypeScript to find the exact shape inside a union. Types like this are called *discriminated union types*, and they're a safe way to move around in TypeScript's type space.

Fixating Value Types

Discriminating unions are a wonderful tool when you want to steer your control flow in the right direction. But it comes with some gotchas when you rely heavily on type inference (which you should).

Let's define a conference object outside of what we get from the back end.

```
const script19 = {
  title: 'ScriptConf',
  date: new Date('2019-10-25'),
  capacity: 300,
  rsvp: 289,
  description: 'The feel-good JS conference',
  kind: 'conference',
  price: 129,
  location: 'Central Linz',
  talks: [{
    speaker: 'Vitaly Friedman',
    title: 'Designing with Privacy in mind',
    abstract: '...'
  }]
};
```

By our type signature, this would be a perfectly fine value of the type `TechEvent` (or `Conference`). However, once we pass

this value to the function `getEventTeaser`, TypeScript will hit us with red squiggly lines.

```
getEventTeaser(script19)
```

According to TypeScript, the types of `script19` and `TechEvent` are incompatible. The problem lies in type inference. The moment we assign this value to the `script19` variable, TypeScript tries to guess the correct type of each property value, and aims for the set it can be most sure will work. As with `const` objects, all properties are still variable, and inferred types are mostly strings and numbers for simple properties.

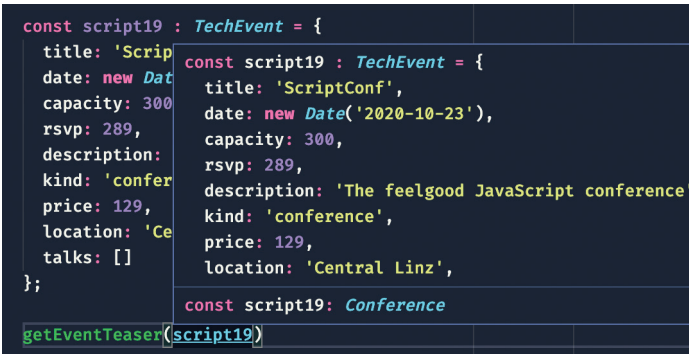
This means the property `kind` in `script19` will not be inferred as `'conference'` but as `string`. And `string` is a much wider set of values than `'conference'`. For this to work, we need to tell TypeScript again that we are looking for the value type, not for its superset of types. We have a couple of possibilities to do that.

First, let's do a left-hand side type annotation.

```
const script19: TechEvent = {
  // All the properties from before ...
}
```

With that, TypeScript does a type check right at the assignment. This way, the value `'conference'` for `kind` will be seen as the annotated value type instead of the much wider `string`.

Not only that, but TypeScript will also understand which subtype of the discriminated type union we are dealing with. If you hover over `script19`, you'll see that TypeScript will correctly understand this value as `Conference`.



Declared as TechEvent, understood as Conference.

But we lose some of the conveniences we get when we rely on type inference. Most of all, we lose the ability to leverage structural typing and work freely with objects that just need to be compatible with types rather than explicitly be of a certain shape.

For scenarios like that, we can fixate certain properties by doing type casts. One way would be to cast the type of property `kind` specifically to the value type:

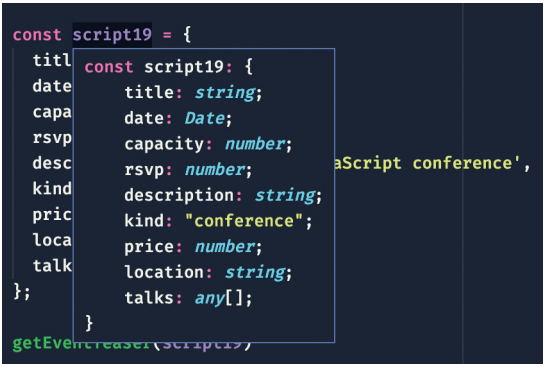
```
const script19 = {
  title: 'ScriptConf',
  date: new Date('2019-10-25'),
  capacity: 300,
  rsvp: 289,
  description: 'The feelgood JS conference',
-  kind: 'conference',
+  kind: 'conference' as 'conference',
  price: 129,
  location: 'Central Linz',
  talks: [{
    speaker: 'Vitaly Friedman',
    title: 'Designing with Privacy in Mind',
    abstract: '...'
  }]
};
```

That will work, but we lose some type safety as we could also cast `'meetup'` as `'conference'`. Suddenly, we again don't know which types we are dealing with, and this is something we want to avoid.

Much better is to tell TypeScript that we want to see this value in its `const` context:

```
const script19 = {
  title: 'ScriptConf',
  date: new Date('2019-10-25'),
  capacity: 300,
  rsvp: 289,
  description: 'The feelgood JS conference',
-  kind: 'conference',
+  kind: 'conference' as const,
  price: 129,
  location: 'Central Linz',
  talks: [{
    speaker: 'Vitaly Friedman',
    title: 'Designing with Privacy in mind',
    abstract: '...'
  }]
};
```

This works just like assigning a primitive value to a `const` and fixate its value type.



What we get with `as const`.

You can apply `const` context events to objects, casting all properties to their value types, effectively creating a value type of an entire object. As a side effect, the whole object becomes read-only.

Lesson 25: Dynamic Unions

Consider the following function. We get a list of tech events and want to filter them by a specific event type:

```
type EventKind =
  'conference' | 'webinar' | 'meetup'

function filterByKind(
  list: TechEvent[],
  kind: EventKind
): TechEvent[] {
  return list.filter(el => el.kind === kind)
}
```

This function takes two arguments: `list`, the original event list; and `kind`, the kind we want to filter by. We return a new list of tech events. We make use of two types to improve type safety. One is `TechEvent`, which we used a lot in the last lessons.

The other one is `EventKind`, a union of all available value types for the property `kind`. With that union in place,

we are allowed to only filter by the kinds of event listed in that union:

```
// A list of tech events we get from a back end
declare const eventList: TechEvent[]

filterByKind(eventList, 'conference') // OK!
filterByKind(eventList, 'webinar') // OK!
filterByKind(eventList, 'meetup') // OK!

// 'concert' is not part of EventKind
filterByKind(eventList, 'concert') // Bang!
```

This is a tremendous improvement for developer experience, but has some pitfalls when our data is changing.

Lookup Types

What if we get another event type to the existing list of event types, called `Hackathon`? A live, in-person coding event that might cost something but has no talks.

Let's define the new type:

```
type Hackathon = TechEventBase & {
  location: string,
  price?: number,
  kind: 'hackathon'
}
```

And add `Hackathon` to the union of TechEvents:

```
type TechEvent =
  Conference | Webinar | Meetup | Hackathon
```

Immediately, we get a disconnect between `EventKind` and `TechEvent`. We can't filter by `'hackathon'` even though it should be possible.

```
// This should be possible
filterByKind(eventList, 'hackathon') // Error
```

One way to change this would be to adapt `EventKind` every time we change `TechEvent`. But this is a lot of effort, especially with growing or changing lists of data. What if, all of a sudden, in-person conferences are not a thing anymore?

We want to keep the changes we make to our types as minimal as possible. For that, we need to create a connection between `EventKind` and `TechEvent`.

You might have noticed that object types have a similar structure to JavaScript objects. It turns out we have similar operators on object types as well.

Just like we can access the property of an object by indexing it, we can access the type of a property by using the right index:

```
declare const event: TechEvent
// Accessing the kind property via the index
// operator
console.log(event['kind'])

// Doing the same thing on a type level
type EventKind = TechEvent['kind']
// EventKind is now
// 'conference' | 'webinar' | 'meetup' | 'hackathon'
```

Since the union of `TechEvent` already combines all possible values of property types into unions, we don't need to define `EventKind` on our own anymore. Types like this are called *index access types* or *lookup types*.

With lookup types we create our own system of connected types that produce red squiggly lines everywhere we didn't expect them, acting as a safeguard for our own, ever-changing work.

Mapped Types

Speaking of dynamically generated types, let's look at a function that groups events by their kind.

```
type GroupedEvents = {
  conference: TechEvent[],
  meetup: TechEvent[],
  webinar: TechEvent[],
  hackathon: TechEvent[]
}

function groupEvents(
  events: TechEvent[]
): GroupedEvents {
  const grouped = {
    conference: [],
    meetup: [],
    webinar: [],
    hackathon: []
  };
  events.forEach(el => {
    grouped[el.kind].push(el)
  })
  return grouped
}
```

The function creates a map, and then stores the original list of tech events in a new order, based on the event kind. Again, we face a similar problem as before. The type `GroupedEvents` is manually maintained. We see that we have four different keys based on the events that we work with, and the moment the original `TechEvent` union changes, we would have to maintain this type as well.

Thankfully, TypeScript has a tool for situations like this. With TypeScript we can create object types by running over a set of value types to generate property keys, and assigning them a specific type.

In our case, we want the keys `hackathon`, `webinar`, `meetup`, and `conference` to be generated automatically and mapped to a `TechEvent` list by running over `EventKind`:

```
type GroupedEvents = {
  [Kind in EventKind]: TechEvent[]
}
```

We call this kind of type *mapped type*. Rather than having clear property names, they use brackets to indicate a placeholder for eventual property keys. In our example, the property keys are generated by looping over the union type `EventKind`. To visualize how this works, let's expand the mapped type ourselves in a couple of steps:

```
// 1. The original declaration
type GroupedEvents = {
  [Kind in EventKind]: TechEvent[]
}

// 2. Resolving the type alias.
// We suddenly get a connection to tech event
type GroupedEvents = {
```

```
  [Kind in TechEvent['kind']]: TechEvent[]
}

// 3. Resolving the union
type GroupedEvents = {
  [Kind in 'webinar' | 'conference'
    | 'meetup' | 'hackathon']: TechEvent[]
}

// 4. Extrapolating keys
type GroupedEvents = {
  webinar: TechEvent[],
  conference: TechEvent[],
  meetup: TechEvent[],
  hackathon: TechEvent[],
}
```

Just like we get from our original type! Mapped types are not only a convenience that allows us to write a lot less and get the same kind of tooling. We also create an elaborate network of connected type information that allows us to catch errors the very moment our data changes.

The moment we add another kind of event to our list of tech events, `EventKind` gets an automatic update and we get more information for `filterByKind`. We also know that we have another entry in `GroupedEvents`, and the function `groupEvents` won't compile because the return

value lacks a key. And we get all these benefits at no extra cost. We just have to be clear with our types and create the necessary connections.

Remember, type maintenance is a potential source of errors. Dynamically updating types helps.

Lesson 26: Object Keys and Type Predicates

Our website not only lists events of different kinds – it also allows users to maintain lists of events they're interested in. For users, events can have different states:

1. Users can be *watching* events they're interested in. They can keep up to date on speaker announcements and more.
2. Users can be actively *subscribed* to events, meaning that they either plan to attend or have already paid the fee. For that, they *responded* to the event.
3. Users can have *attended* past events. They want to keep track of video recordings, feedback, and slides.
4. Users can have *signed out* of events, meaning they were either subscribed to an event but changed their mind, or they just don't want to see that event in their

lists anymore. Our application keeps track of those events as well.

As always, we want to model our data first. As we don't want to change our existing types, but want a quick way to access all four categories, we create another object that serves as a map to each category. The type for this object looks like this:

```
type UserEvents = {
  watching: TechEvent[],
  rvsp: TechEvent[],
  attended: TechEvent[],
  signedout: TechEvent[],
}
```

Now for some operations on this object.

keyof

We want to give users the option to filter their events. First by category: `watching`, `rsvp`, `attended`, and `signedout`; second – and optionally – by the kind of event: conference, meetup, webinar, or hackathon. The function we want to create accepts three arguments:

1. The `userEventList` we want to filter.
2. The `category` we want to select. This matches one of the keys of the `userEventList` object.

3. Optionally, a string of the set `EventKind` that allows us to filter even further.

The first filter operation is quite simple. We want to access one of the lists via the index access operator; for example, `userEventList['watching']`. So for the type of the `category` we create a union type that includes all keys of `userEventList`.

```
type UserEventCategory =
  'watching' | 'rsvp' | 'attended' | 'signedoff'

function filterUserEvent(
  userEventList: UserEvents,
  category: UserEventCategory,
  filterKind?: EventKind
) {
  const filteredList = userEventList[category]
  if (filterKind) {
    return filteredList.filter(event =>
      event.kind === filterKind)
  }
  return filteredList
}
```

This works, but we face the same problems as we did in the previous lesson: we're maintaining types manually, which is prone to errors and typos. Problems of that kind that are hard to catch. Perhaps you didn't notice I made a mistake by using the value type `signedoff` in `UserEventCategory`, which isn't a key in `UserEvents`. That would be `signedout`.

We want to create types like this dynamically, and TypeScript has an operator for that. With `keyof` we can get the object keys of *every* type we define. And I mean *every*. We can use `keyof` even with value types of the string set and get all string functions. Or with an array and get all array operators:

```
// 'speaker' | 'title' | 'abstract'
type TalkProperties = keyof Talk

// number | 'toString' | 'charAt' | ...
type StringKeys = keyof 'speaker'

// number | 'length' | 'pop' | 'push' | ...
type ArrayKeys = keyof []
```

The result is a union type of value types. We want the keys of our `UserEvents`, so this is what we do:

```
function filterUserEvent(
  userEventList: UserEvents,
  category: keyof UserEvents,
  filterKind?: EventKind
) {
  const filteredList = userEventList[category]
  if (filterKind) {
    return filteredList.filter(event =>
      event.kind === filterKind)
```

```
  }
  return filteredList
}
```

The moment we update our `UserEvent` type, we also know which keys we have to expect. So if we remove something, instances where a removed key is used get red squiggly lines. If we add another key, TypeScript will give us proper autocomplete for it.

Type Predicates

Let's assume that `filterUserEvents` is not only within our application, but also available outside. Other developer teams in our organisation can access the function, and they might not use TypeScript to get their job done. For them, we want to catch some possible errors up front, while still retaining our type safety.

From both filter operations, the `category` filter is the problematic one, as it could access a key that is not available in `userEventList`. To keep it type-safe for us, and more flexible to the outside, we accept that `category` is not a subset of string, but the whole set of strings:

```
function filterUserEvent(
  list: UserEvents,
  category: string,
  filterKind?: EventKind
) {
  // ... tbd
}
```

But before we access the category, we want to check if this is a valid key in our list. For that, we create a helper function called `isUserEventListCategory`:

```
function isUserEventListCategory(
  list: UserEvents,
  category: string
) {
  return Object.keys(list).includes(category)
}
```

and apply this check to our function:

```
function filterUserEvent(
  list: UserEvents,
  category: string,
  filterKind?: EventKind
) {
  if(isUserEventListCategory(list, category)) {
    const filteredList = list[category]
```

```
    if (filterKind) {
      return filteredList.filter(event =>
        event.kind === filterKind)
    }
    return filteredList
  }
  return list
}
```

This is enough safety to not crash the program if we get input that doesn't work for us. But TypeScript (especially in strict mode) is not happy with that. We lose all connections to `UserEvents`, and `category` is still a string. On a type level, how can we be sure that we access the right properties?

This is where *type predicates* come in. Type predicates are a way to add more information to control flow analysis. We can extend the possibilities of narrowing down by telling TypeScript that if we do a certain check, we can be sure our variables are of a certain type:

```
function isUserEventListCategory(
  list: UserEvents,
  category: string
): category is keyof UserEvents { // The type
predicate
  return Object.keys(list).includes(category)
}
```

Type predicates work with functions that return a Boolean. If this function evaluates to true, we can be sure that `category` is a key of `UserEvents`. This means that in the true branch of the if statement, TypeScript knows the type better. We narrowed down the set of `string` to a smaller set `keyof UserEvents`.

Lesson 27: Down at the Bottom: never

With all that widening and narrowing of sets, even down to single values being a type, we have to ask ourselves: can we get even narrower?

Yes, we can. There's one type that's at the very bottom of the type hierarchy. One type that is an even smaller set than a set with one value. The type without values. The empty set: `never`.

never in Control Flow Analysis

`never` behaves pretty much like the anti-type of `any`. Whereas `any` accepts all values and all operations on those values, `never` doesn't accept a single value at all. It's impossible to assign a value and, of course, there are no operations we

can do on a type that is `never`. So what does a type with no values feel like when we are working with it?

We briefly touched on this already; it was hidden in plain sight. Let's go back to lesson 24 and remember what we did when writing the `getEventTeaser` function, now with the `Hackathon` type included:

```
function getEventTeaser(event: TechEvent) {
  switch(event.kind) {
    case 'conference':
      return `${event.title} (Conference), ` +
        `priced at ${event.price} USD'
    case 'meetup':
      return `${event.title} (Meetup), ` +
        `hosted at ${event.location}`
    case 'webinar':
      return `${event.title} (Webinar), ` +
        `available online at ${event.url}`
    case 'hackathon':
      return `${event.title} (Hackathon)`
    default:
      throw new Error('Not sure what to do with that!')
  }
}
```

This `switch` statement runs through all the value types within the `EventKind` union type: `'conference' | 'meetup' | 'webinar' | 'hackathon'`. With every `case` state-

ment in our `switch`, TypeScript knows to take one value type away from this list. After we've checked for `'conference'`, it can't be checked again later on.

Once this list is exhausted, we have no more values left in our set. The list is empty. This is the `default` branch in our `switch` statement.

But, if we checked for all values in our list, why would we run into a `default` branch anyway? Wouldn't that be erroneous behaviour?

Exactly! This is highly erroneous, as we indicate by throwing a new error right away! Running into the `default` branch can never happen. *Never!*

There it was, the *never* word. So this is what type `never` is all about. It indicates the cases that aren't supposed to happen, telling us that we should be very careful as our variables probably don't contain the values we expect.

If you take the example above, enter `event` in the first line of the `default` branch and hover over it, TypeScript will show you exactly that.

```
function getEventTeaser(event: TechEvent) {
  switch(event.kind) {
    case 'conference':
      return `${event.title} (Conference)`
    case 'meetup':
      return `${event.title} (Meetup)`
    case 'webinar':
      return `${event.title} (Webinar)`
    ca
                                    ckathon)`
    de
      event
      throw new Error('No idea what to do')
  }
}
```

```
event: TechEvent
(parameter) event: never
```

The list is exhausted, `event` *is* `never`.

Any operation on `event`, other than being part of an error thrown, will cause compiler errors. This is a situation that should never happen at all!

Preparing for Dynamic Updates

Right now, our `getEventTeaser` function deals with all entries from `EventKind`. In the case of a value coming in that isn't part of the union type, we throw an error. This is great, but only works if we handle all possible cases.

What if we haven't exhausted our entire list yet? Let's remove `'hackathon'` for now:

```
function getEventTeaser(event: TechEvent) {
  switch(event.kind) {
    case 'conference':
      return `${event.title} (Conference), ` +
        `priced at ${event.price} USD`
    case 'meetup':
      return `${event.title} (Meetup), `+
        `hosted at ${event.location}`
    case 'webinar':
      return `${event.title} (Webinar), ` +
        `available online at ${event.url}`
    default:
      throw new Error('Not sure what to do with
that!')
  }
}
```

In the default branch, `event.kind` is now `'hackathon'`, but we aren't dealing with it – we just throw an error. This is somewhat right as *we are not sure what to do with that*, but it would be a lot nicer if TypeScript alerted us that we forgot something. We want to exhaust our entire list, after all.

For that, we want to make sure that at the end of a long `switch-case` statement, or in `else` branches that shouldn't occur, the type of `event` is definitely `never`. Let's create a

utility function that throws the error. But instead of sending just a message, we also want to send the culprit that eventually caused that error. Clue: the type of this culprit is `never`.

```
function neverError(
  message: string,
  token: never // The culprit
) {
  return new Error(
    `${message}. ${token} should not exist`
  )
}
```

We substitute the `neverError` function with the actual error throwing in our `switch-case` statement:

```
function getEventTeaser(event: TechEvent) {
  switch(event.kind) {
    case 'conference':
      return `${event.title} (Conference), ` +
        `priced at ${event.price} USD`
    case 'meetup':
      return `${event.title} (Meetup), ` +
        `hosted at ${event.location}`
    case 'webinar':
      return '${event.title} (Webinar), ` +
        'available online at ${event.url}`
    default:
      throw neverError(
        'Not sure what to do with that',
```

```
        event
      )
  }
}
```

And immediately TypeScript's type checking powers kick in. At this point, `event` could potentially be a hackathon. We're just not dealing with that. TypeScript gives us a red squiggly and tells us that we can't pass some value to a function that expects `never`.

After we add `'hackathon'` to the list again, TypeScript will compile again, and all our exhaustive checks are complete.

```
function getEventTeaser(event: TechEvent) {
  switch(event.kind) {
    case 'conference':
      return `${event.title} (Conference), ` +
        `priced at ${event.price} USD`
    case 'meetup':
      return `${event.title} (Meetup), ` +
        `hosted at ${event.location}`
    case 'webinar':
      return `${event.title} (Webinar), ` +
        `available online at ${event.url}`
    case 'hackathon':
      return `even that: ${event.title}`
    default:
      throw neverError(
```

```
        'Not sure what to do with that',
        event // No complaints
      )
  }
}
```

With `never` we get a safeguard that can be used for situations that could occur, but should never occur. Especially when dealing with sets of values that get wider and narrower as we code our applications.

`never` is the bottom type of all other types, and will be a handy tool in the next chapters.

Lesson 28: undefined and null

Before we close this chapter, we have to talk about two special value types that you will catch sooner or later in your applications: `null` and `undefined`.

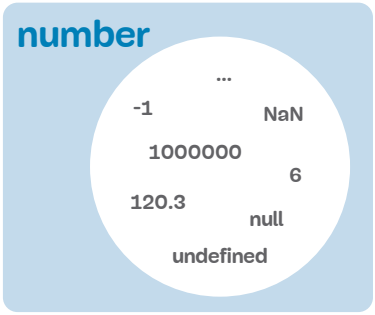
Both `null` and `undefined` denote the absence of a value. `undefined` tells us that a variable or property has been declared, but no value has been assigned. `null`, on the other hand, is an *empty value* that can be assigned to clear a variable or property.

Both values are known as *bottom values*, values that have no actual value.

Douglas Crockford once said[25] that there is a lot of discussion in the programming languages community about whether a programming language should even have bottom values. Nobody has the opinion that there need to be two of them.

undefined and null in the Type Space

`undefined` and `null` are somewhat special in TypeScript. Both values are regularly part of each set of types.



The type number with `undefined` *and* `null`.

This is because JavaScript behaves that way. The moment we declare a variable, it is set to `undefined`. Programatically, we can set variables to `null` or `undefined`. But this brings along some problems.

25 https://smashed.by/crockford

Let's look at this simple example:

```
// Let's define a number variable
let age: number

// I'm getting one year older!
age = age + 1
```

This is valid TypeScript code. We declare a number, and add another number value to it. The problem is that this brings us values we would not expect.

The result of this operation is `NaN`, because we are adding `1` to `undefined`. Technically, the result is again of type `number`, just not what we expected!

It can get worse. Let's go back to our tech event example. We want to create an HTML representation of one of our events and append it to a list of elements. We create a function that runs over the common properties and returns a string:

```
function getTeaserHTML(event: TechEvent) {
  return `<h2>${event.title}</h2>
    <p>
      ${event.description}
    </p>`
}
```

We use this function to create a list element, which we can add to our list of events:

```
function getTeaserListElement(event: TechEvent) {
  const content = getTeaserHTML(event)
  const element = document.createElement('li')
  element.classList.add('teaser-card')
  element.innerHTML = content
  return element
}
```

A bit rough, but it does the trick. Now, let's add this element to a list of existing elements:

```
function appendEventToList(event: TechEvent) {
  const list = document.querySelector('#event-list')
  const element = getTeaserListElement(event)
  list.append(element)
}
```

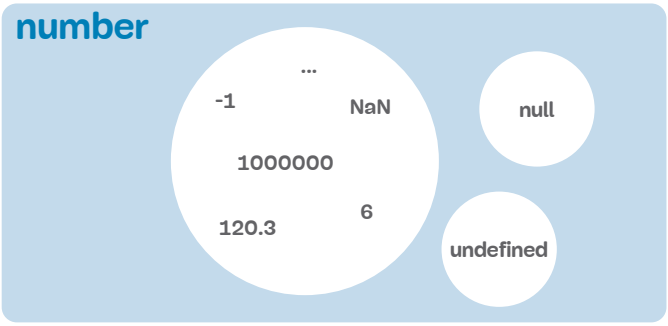
And here's the problem: we have to be very sure that an element with the ID `event-list` exists in our HTML. Otherwise `document.querySelector` returns `null`, and appending the list will break the application.

Strict null Checks

With `null` being part of all types, the code above is both valid and highly toxic. A simple change in our markup and the whole application breaks. We need a way to make sure that the result of `document.querySelector` is actually available and not `null`.

Of course, we can do `null` checks or use the fancy "Elvis" operator (`?.` also known as optional chaining[26]), but wouldn't it be great if TypeScript told us actively that we should do so?

There is a way. In your *tsconfig.json* we can activate the option `strictNullChecks` (which is part of strict mode). Once we activate this option, all **nullish** values are excluded from our types.



The type number with strict null checks.

26 https://smashed.by/optionalchaining

With `null` and `undefined` not being part of the actual type set, this piece of code will cause an error during compile time:

```
let age: number
age = age + 1
```

`age` is not defined after all! But `strictNullChecks` does not change how `document.querySelector` works. The result can still be `null`. But the return type of `document.querySelector` is `Element | null`, a union type with the nullish value! And this makes TypeScript immediately throw a red squiggly at us:

```
function appendEventToList(event: TechEvent) {
  const list = document.querySelector('#event-list')
  const element = getTeaserListElement(event)
  list.append(element)
}
```

`list` is probably `null`. How right TypeScript is. A quick `null` check (the Elvis operator[27] dancing in front of us) does the trick and makes our code a lot safer:

```
function appendEventToList(event: TechEvent) {
  const list = document.querySelector('#event-list')
  const element = getTeaserListElement(event)
```

27 https://smashed.by/elvis

```
  list?.append(element) // Optional chaining / Null
check
}
```

Typescript goes a little bit further even. With `strict NullChecks` enabled, we not only have to check for nullish values, we are also not allowed to assign `undefined` or `null` to variables and properties. Both values are removed from all types, so an assignment of that kind is forbidden.

There are situations where we need to work with either `undefined` or `null`. To bring one (or both) values back into the mix, we have to add them to a union; for example, `string | undefined`. This makes adding nullish values explicit, and we have to check for their existence.

```
type Talk = {
  title: string,
  speaker: string,
  abstract: string | undefined
}
```

Another way to add `undefined` is to make properties of an object optional. Optional properties have to be checked for as well, but without us maintaining too many types.

```
type Talk = {
  title: string,
  speaker: string,
  abstract?: string
}
```

In any case, like Douglas Crockford said, why should we need two nullish values? If you must use one, stick with one of them.

Recap

This chapter was all about type hierarchies, set theory, top and bottom types, and nullish values that can break our programs. Everything we learned in the scope of union and intersection types is crucial to everything that's coming up. Once you learn how to move around in the type space, TypeScript has so much to offer you.

1. We learned about union and intersection types, and how we can model data that can take different shapes.
2. We also learned how union and intersection types work within the type space. We also learned about discriminating unions and value types.

3. We learned about `const` context, and found ways to dynamically create other types through lookup and mapped types.
4. We built our own type predicates as custom type guards.
5. The bottom type `never` is great for exhaustive checks within `switch` or `if-else` statements.
6. Last, but not least, we dealt with `null` and `undefined` and got pretty much rid of them.

One thing that is now second nature to us is widening and narrowing types. We can go from the all-encompassing `any` down to the type with no values, `never`. We can freely move around in the type space for all types we know of. Now let's learn what to do with types whose shapes we don't know.

Interlude: Tuple Types

We traversed the whole type spectrum of primitive types and object types, but there's one detail we've left out: arrays and their subtypes. Consider this function signature:

```
declare function useToggleState(id: number):
  { state: boolean, updateState: () => void };
```

You might see something like this when you use a library like React. It takes one parameter, a number. The name suggests it's an identifier, and it returns an object with the state of our toggle button, and a function to update this state.

When we use this function, we want to use destructuring to have easy access to its properties:

```
const { state, updateState } = useToggleState(1)
```

But what happens if we need to use more than one toggle state at the same time?

```
const { state, updateState } = useToggleState(1)
// Those variables are already declared!
const { state, updateState } = useToggleState(2)
```

Object destructuring lets us go directly to the properties of an object, declaring them as variables. We can use array destructuring to go directly to the indices of an array, declaring them as variables under an entirely new name:

```
const [ first, updateFirst ] = useToggleState(1)
const [ second, updateSecond ] = useToggleState(2)
```

Now we can use `first`, `second` and their state update methods freely in our code. Of course, we would require `useToggleState` to return an array instead.

But how do we type this? We are dealing with two different types. One is Boolean, the other one a function with no parameters and no return value. This is not your average array with a technically endless amount of values of one type.

It's a tuple. While an array is a list of values that can be of any length, we know exactly how many values we get in a tuple. Usually, we also know the type of each element in a tuple.

In TypeScript, we can define tuples. A tuple type for the example above would be

```
declare function useToggleState(id: number):
  [boolean, () => void]
```

Note that we don't define properties, just types. The order in which the types appear is important.

Tuple types are subtypes of arrays, but they can't be inferred. If we use type inference directly on a tuple, we will get the wider `array` type:

```
// tuple is '(string | number)[]'
let tuple = ['Stefan', 38]
```

As with any other value type, declaring a `const` context can infer the types correctly:

```
// tuple is read-only [string, number]
let tuple = ['Stefan', 38] as const
```

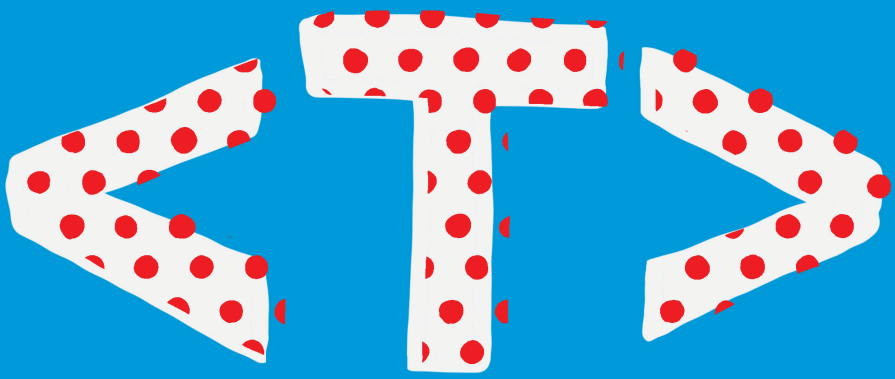
But this makes `tuple` read-only too, so be aware. As with any other subtype, if we declare a narrower type in a function signature or in a type annotation, TypeScript will check against the narrower type instead of the wider, more general type:

```
function useToggleState(id: number):
  [boolean, () => void] {
  let state = false
```

```
  // ... Some magic

  // Type checks!
  return [false, () => { state = !state}]
}
```

Without the return type, TypeScript would assume that we get an array of mixed Boolean and function values.



Chapter Five

GENERICS

Generics

In chapter 4 we learned how to move in the type space, and how we can narrow and widen sets of types. We get more type safety by knowing exactly what we can expect from our software at a particular point in time.

But there are situations where we can't say for sure what awaits us. Situations where we have a notion of what's coming up, but some details remain shrouded in mystery. Still, we want to get type safety and all the nice tooling features TypeScript provides.

Generics offer us a way to prepare for the unknown. They let us define types that describe a certain piece of the type system where the details are filled out later. This is the land where utility functions and utility types are born. To illustrate the following chapter, think of a video player portal that features video streams of different qualities, subtitles in different languages, and user-centric features.

Lesson 29: I Don't Know What I Want, but I Know How to Get It

The infamous line by the Sex Pistols perfectly describes punk of the 1970s, toddlers during tantrums, and generics.

Consider the following data structure for a video that exists in different formats:

```
type VideoFormatURLs = {
  format360p: URL,
  format480p: URL,
  format720p: URL,
  format1080p: URL
}
```

`URL` is the browser's built-in class of URLs. We want to provide an API where developers can load a specific format (using `declare` statements for brevity):

```
declare const videos: VideoFormatURLs
declare function loadFormat(
  format: string
): void
```

To make sure the incoming `format` is a valid key in our data structure, we create a utility function with a type predicate, just like we did in the previous chapter:

```
function isFormatVailable(
  obj: VideoFormatURLs,
  key: string
): key is keyof VideoFormatURLs {
```

```
  return key in obj
}
```

The function works as intended, and in our type space we can narrow down the set of all strings to just the keys of `VideoFormatURLs`:

```
if(isFormatAvailable(videos, format)) {
  // format is now "format360p" | "format480p" |
  //               "format720p" | "format1080p"
  // and index accessing perfectly works:
  videos[format]
}
```

Now, we have a similar situation for loading subtitles. This is our subtitle data structure:

```
type SubtitleURLs = {
  english: URL,
  german: URL,
  french: URL
}
```

And this is the validation function to check if a certain key is available in our subtitles object:

```
function isSubtitleAvailable(
  obj: SubtitleURLs,
  key: string
): key is keyof SubtitleURLs {
  return key in obj
}
```

Wait a minute. This is exactly the same function! In JavaScript, we wouldn't create two of them, as they serve exactly the same purpose and even have exactly the same implementation! But we need two implementations because we want to have type safety, don't we?

Well, here's a rule to live by: if we do something in TypeScript that we wouldn't do like that in JavaScript, we should rethink. TypeScript was designed to provide type safety for almost all JavaScript scenarios. Correctly typing a utility function is definitely one of them.

Enter Generics

Let's take a step back and define a utility function in the way we would in JavaScript, without any types:

```
function isAvailable(
```

```
  obj, key
) {
  return key in obj
}
```

Now, we want to prepare our function for unknown types and still give the correct answer. This is where generics come in. Generic programming is

> *a style of computer programming in which algorithms are written in terms of types-to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)*[28]

This definition includes some crucial information: instead of working with a specific type, we work with a parameter that is then substituted for a specific type. Type parameters are denoted within angle brackets at function heads or class declarations. Let's add one to our `isAvailable` function.

```
function isAvailable<Formats>(
  obj, key
) {
  return key in obj
}
```

28 https://smashed.by/generic

The type `Formats` does not exist in our type declarations, but is a parameter that gets substituted for a real one, like `VideoFormatURLs` or `SubtitleURLs`. However, we can use this type parameter with regular types within our function declaration:

```
function isAvailable<Formats>(
  obj: Formats, key
): key is keyof Formats {
  return key in obj
}
```

If we want to type `key` – which is now implicitly `any` – we would need to use a wider set of possible key types:

```
function isAvailable<Formats>(
  obj: Formats,
  key: string | number | symbol
): key is keyof Formats {
  return key in obj
}
```

This is because our generic type parameter `Formats` doesn't know it can only have string keys; it has to prepare itself for all possible keys. In JavaScript, numbers and symbols are all valid key types. Take an array, for instance; arrays can be seen as objects with number keys.

Generic Annotations and Generic Inference

Now that we've defined our function as a generic function, let's use it. We have two different ways of using the generic function. First, we can explicitly annotate the type we want to substitute:

```
if(isFormatAvailable<VideoFormatURLs>(videos, format))
{
  // ...
}
```

Just like explicit type annotations elsewhere, TypeScript takes this as a given and validates everything else against this type. This means that the moment we specify `VideoFormatURLs` to be the substitute for the type parameter, we have to make sure that the argument `obj` that we pass to the function matches the type `VideoFormatURLs`.

However, it's much more interesting and powerful when we use type inference to substitute our type parameter. TypeScript is capable of inferring the type parameter from actual arguments you pass to a function, which feels much more natural:

```
// An object with video formats
declare const videoFormats: VideoFormatURLs

if(isAvailable(videoFormats, format)) {
  // Inferred type 'VideoFormatURLs'
  // format is now keyof VideoFormatURLs
}
// An object with video formats
declare const subtitles: SubtitleURLs

if(isAvailable(subtitles, language)) {
  // Inferred type 'SubtitleURLs'
  // language is now keyof SubtitleURls
}
```

This is just writing JavaScript.

Generics in the Wild

This was our first self-written, generic function. You might have encountered some generics already. `Promise` is a generic type that pops up the moment you write asynchronous code. The argument in `Promise` gives you the result type:

```
// randomNumber returns a Promise<number>
async function randomNumber() {
  return Math.random()
}
```

Another one is `Array`. We can write array types with an array literal:

```
let anArray: number[]
```

Or we can use the generic

```
let anotherArray: Array<number>
```

Both do the same thing. You might find it a bit more convenient, however, to use the generic type when you deal with union types:

```
let aMixedArray: Array<number | string | boolean>
```

When you work particularly with lots of JavaScript's built-in APIs or browser APIs, you will encounter generics.

Lesson 30: Generic Constraints

Our generic function is already pretty good. We can pass anything from the wide variety of types available and can

pinpoint concrete types once we substitute. When we think of sets, we open up a type for `any`; and then once we substitute, we select a much narrower set.

This can lead to some undesired behavior, unfortunately. Our `isAvailable` type from the last lesson works really well with the object types we defined:

```
function isAvailable<FormatList>(
  obj: FormatList,
  key: string | number | symbol
): key is keyof FormatList {
  return key in obj
}

// An object with video formats
declare const videoFormats: VideoFormatURLs

if(isAvailable(videoFormats, format)) {
  // Inferred type 'VideoFormatURLs'
  // format is now keyof VideoFormatURLs
}
// An object with video formats
declare const subtitles: SubtitleURLs

if(isAvailable(subtitles, language)) {
  // Inferred type 'SubtitleURLs'
  // language is now keyof SubtitleURls
}
```

And also with all other object types that are available – even ones without a concrete type declaration:

```
if(isAvailable({ name: 'Stefan', age: 38}, key)) {
  // key is now “name” | “age”
}
```

But it also works with all non-object types:

```
if(isAvailable('A string', 'length')) {
  // Also strings have methods,
  // like length, indexOf, ...
}

if(isAvailable(1337, aKey)) {
  // Also numbers have methods
  // aKey is now everything number has to offer
}
```

It also works with arrays, such that the key can be the entire set of numbers as well as array functions like `map`, `forEach`, and so on.

While this is cool, as it makes our types even more compatible, it can lead to undesired behavior if we only want to check objects. Thankfully, TypeScript has a way to deal with situations like this.

Defining Boundaries

As we explained initially, type parameters of generics cover the entire set of types: therefore, `any`. By substituting with a specific type, the type set gets narrower and clearer.

However, there's the possibility to define boundaries, or subsets of the type space. This makes generic type parameters a little bit narrower before they're substituted by real types. We get information up front if we pass an object that shouldn't be passed.

To define generic subsets, TypeScript uses the `extends` keyword. We check if a generic type parameter extends a specific subset of types. If we only want to pass objects, we can extend from the type object:

```
function isAvailable<FormatList extends object>(
  obj: FormatList,
  key: string
): key is keyof FormatList {
  return key in obj
}
```

With `<FormatList extends object>`, we tell TypeScript that the argument we pass needs to be at least an object. All primitive types and even arrays are excluded.

```
isAvailable('A string', 'length')
```

Red squigglies where they are supposed to be.

Index Types

Let's define a function that loads a file, either a video in a specific format, or a subtitle in a specific language. Again, we start with the raw JavaScript function (just the head for brevity):

```
function loadFile(fileFormats, format) {
  // Implement
}
```

When we add types, we would do something similar, as with the `isAvailable` function:

```
function loadFile<Formats extends object>(
  fileFormats: Formats,
  format: string
) {
  // You know
}
```

We can go even further. When you look at both our format definitions, you'll recognize another common feature.

```
type VideoFormatURLs = {
  format360p: URL,
  format480p: URL,
  format720p: URL,
  format1080p: URL
}

type SubtitleURLs = {
  english: URL,
  german: URL,
  french: URL
}
```

That's right: all properties are of type `URL`. Another format would most likely cause an error when used with the `loadFiles` function.

We would need a constraint to ensure that we only pass compatible objects, where we don't know the properties themselves, but only know that every property is of type `URL`.

Index types, like we briefly saw in chapter 4, are perfect for this. Below is an index type that we've met before, iter-

ating over a set of unions, and in this case allowing any value for each property:

```
type PossibleKeys = 'meetup' | 'conference'
  'hackathon' | 'webinar'

type Groups = {
  [k in PossibleKeys]: any
}
```

Index types don't define specific property keys. They just define a set of keys they iterate over. We can also accept the entire set of strings as keys.

```
type AnyObject = {
  [k: string]: any
}
```

Now that we accept all property keys of type `string`, we can explicitly say that the type of each property needs to be `URL`:

```
type URLList = {
  [k: string]: URL
}
```

A perfect shape that includes `VideoFormatURLs` as well as `SubtitleURLs`. And basically any other list of URLs! Therefore, also a perfect constraint for our generic type parameter:

```
type URLList = {
  [k: string]: URL
}

function loadFile<Formats extends URLList>(
  fileFormats: Formats,
  format: string
) {
  // The real work ahead
}
```

With that, every object we pass that doesn't give an object with URLs is going to create beautiful, red, squiggly lines in our editor.

Lesson 31: Working with Keys

We defined an object that allows for any key of type `string`, as long as the type of each property is `URL`. With that, we already know that we only can pass objects that have the correct shape without the compiler complaining.

However, when we select the right format, we still can pass every string to the function, even though the format might not exist.

```
declare const videos: VideoFormatURLs

loadFile(videos, 'format4k')
 // 4K not available
 // TypeScript doesn't squiggle
```

Of course, we can do better.

Related Type Parameters

We only want to pass keys as the second argument that are actually available in the object. In a non-generic world, we would do something like this:

```
function loadVideoFormat(
  fileFormats: VideoFormatURLs,
  format: keyof VideoFormatURLs
) {
  // You know
}
```

And the same applies to generic type parameters:

```
type URLObject = {
  [k: string]: URL
}

function loadFile<Formats extends URLObject>(
  fileFormats: Formats,
  format: keyof Formats
) {
  // The real work ahead
}
```

This already gives us great tooling. Now we can only enter keys which are part of the object we pass as the first parameter:

```
loadFile(video, 'format1080p') // thumbs up!

// 'format4k' is not available
loadFile(video, 'format4k')
```

`keyof Formats`, when substituted with `VideoFormatURLs` yields `"format360p" | "format480p" | "format720p" | "format1080p"`. The format we pass for the second argument needs to be within this union type.

Let's take a look at the function body and do a very straightforward implementation. We access the URL, fetch some data from it, and return an object that tells us what format we loaded, and if loading was successful. An actual implementation would have much more detail, but this is all we need to see what's happening on a type level.

As we are using the `async fetch` function, we are transforming `loadFile` to be an `async` function as well.

```
async function loadFile<Formats extends URLObject>(
  fileFormats: Formats,
  format: keyof Formats
) {
  // Fetch the data
  const data = await fetch(fileFormats[format].href)
  return {
    // Return the format
    format,
    // and see if we get an OK response
    loaded: data.response === 200
  }
}
```

Let's see what we get in return. Thanks to type inference, the return type of `loadFile` is `Promise<{ format: keyof Formats, loaded: boolean }>`. `Promise` is a generic type,

which shouldn't come as any surprise by now. And the property `format` in our return value is the generic type parameter we defined in our function.

Let's use our function with substitutes.

```
const result = await loadFile(videos, "format1080p")
```

`await` is unwrapping the `Promise<>`, so we can see that the actual return values from `loadFile`. `result` is of type `{ format: "format360p" | "format480p" | "format720p" | "format1080p", loading: boolean }`. As we expect, we get `keyof VideoFormatURLs` as return.

But shouldn't we know more? We are explicitly passing `"format1080p"` as second argument. We've already narrowed down the union through usage to a single value type. Why can't `result` be of type `{ format: "format1080p", loading: boolean }`?

We can achieve this by adding a second type parameter to our generic declaration, one that shows the relationship with the first, but works as its own type once we declare it, like this:

```
function loadFile<
  Formats extends URLObject,
```

```
  Key extends keyof Formats
>(fileFormats: Formats, format: Key) {
  const data = await fetch(fileFormats[format].href)
  return {
    format,
    loaded: data.response === 200
  }
}
```

The second type parameter `Key` is a subtype of `keyof Formats`, the first type parameter. The interesting part now happens when we start subtituting:

```
loadFile(video, 'format1080p') // OK!
```

`video` is of type `VideoFormatURLs`. `VideoFormatURLs` is a subtype of `URLObject`, so the type check passes and `Formats` can be substituted. Now `Key` needs to be a subtype of `keyof Formats`. `"format1080p"` is a subtype of `keyof Formats`, so the type check passes, and `Key` can be substituted.

Now we've locked in and substituted two types:

7. `Formats` is `VideoFormatURLs`
8. `Key` is `"format1080p"`

That's right. Now that we've passed a literal string, the type parameter takes the literal, the value type, which means that once we execute this function and look at the result, we can be sure that `result.format` is `"format1080p"`:

```
const result = await loadFile(videos, “format1080p”)

if(result.format !== “format1080p”) {
  // result.format is now never!
  throw new Error(“Your implementation is wrong”)
}
```

To make sure that we're also implementing the right thing, we define a return type for the `loadFile` function where we expect the `Key` type to appear.

```
type URLObject = {
  [k: string]: URL
}

type Loaded<Key> = {
  format: Key,
  loaded: boolean
}

async function loadFile<
  Formats extends URLObject,
  Key extends keyof Formats
>(fileFormats: Formats, format: Key):
Promise<Loaded<Key>> {
```

```
  const data = await fetch(fileFormats[format].href)
  return {
    format,
    loaded: data.response === 200
  }
}
```

All wrapped in a generic `Promise` as we are async.

Lesson 32: Generic Mapped Types

TypeScript has a couple of helper types that can be used for what we did manually. They might come in handy when we start creating advanced types. Let's look at `Record` and `Pick`. Both are mapped types with generics.

Pick

`Pick<O, K>` creates a new object with selected property keys `K` of object `O`. It is defined as

```
type Pick<
  O,
  K extends keyof O
```

```
> = {
  [P in K]: O[P];
}
```

`[P in K]` runs over all value types in the union `K`, which is all keys of `O`. `O[P]` is an *indexed access type*. It's like indexing an object, but retrieving a type. This allows us to define a union of keys that are part of an original object type, and select those keys and their types from the original object.

For example, this would be a type with all HD videos

```
type HD = Pick<
  VideoFormatURLs,
  'format1080p' | 'format720p'
>

// Equivalent to

type HD = {
  format1080p: URL,
  format720p: URL
}
```

The `Pick` helper type's most obvious use is the `pick` utility function available from libraries like Lodash. But it can

be helpful in other scenarios as well. We'll look at some of those in later chapters.

Record

`Record<K, T>` creates an object type where all types in `T` get the type `K`. Like a dictionary.

It is defined as

```
type Record<
  K extends string | number | symbol,
  T
> = {
  [P in K]: T
}
```

Note that `K` is a subtype of `string | number | symbol`. We met this trio earlier on, as they are the allowed types for object keys.

`URLObject` from the previous lesson would be defined as

```
type URLObject = Record<string, URL>
```

`Record` is a neat shorthand if we need to create an object type on the fly.

Mapped and Indexed Access Types

Let's say that our video platform, while allowing for all four kinds of video resolution to be uploaded, doesn't require all four of them. We require at least one format.

Modeling this situation is easily done with union types:

```
type Format360 = {
  format360p: URL
}

type Format480 = {
  format480p: URL
}

type Format720 = {
  format720p: URL
}

type Format1080 = {
  format1080p: URL
}

type AvailableFormats =
  Format360 | Format480 | Format720 | Format1080
```

```
const hq: AvailableFormats = {
  format720p: new URL('...'),
  format1080p: new URL('...')
} // OK!
const lofi: AvailableFormats = {
  format360p: new URL('...'),
  format480p: new URL('...')
} // OK!
```

With union types, we only need to fulfill the contract of one union constituent. This makes it great if we need a minimum of one random property set, and all others are optional.

But – you guessed it – it would require us to maintain a second set of types. We don't want to redefine `VideoFormatURLs`, as the type is necessary for certain functionality in our app. We just want to have `VideoFormatURLs` but split into unions. Let's build a helper, called `Split`.

The goal is to create a union type. To make it easier, we start with a concrete type and work with the substitution later. So what do we already know?

First, we know `keyof VideoFormatURLs` creates a union of all keys of `VideoFormatURLs`.

```
type Split = keyof VideoFormatURLs

// Equivalent to
type Split =
  “format360p” | “format480p” |
  “format720p” | “format1080p”
```

We also know that a mapped type runs over all keys and creates a new object with those keys. The following example creates the same type as `VideoFormatURLs`, but with the key also being the value:

```
type Split = {
  [P in keyof VideoFormatURLs]: P
}

// Equivalent to
type Split = {
  format360p: “format360p”,
  format480p: "format480p”,
  format720p: “format720p”,
  format1080p: “format1080p”
}
```

Now we can access the values of this type again by using the *indexed access operator*. If we access by the union of keys of `VideoFormatURLs`, we get a union of the values.

```
type Split = {
  [P in keyof VideoFormatURLs]: P
}[keyof VideoFormatURLs]

// Equivalent to
type Split =
  “format360p” | “format480p” |
  “format720p” | “format1080p”
```

This looks exactly like the first step, but it's fundamentally different. Instead of getting the left side of an object type – the property keys – in union, we get the right side of an object type – the property types – in union.

So the only thing we have to do is to get the values right, and we have the union we envisioned. Enter `Record`. A `Record<P, VideoFormatURLs[P]` gives us an object with the property `P` we get from the key union, and we're accessing the corresponding type from the property key.

```
type Split = {
  [P in keyof VideoFormatURLs]
    : Record<P, VideoFormatURLs[P]>
}[keyof VideoFormatURLs]

// Equivalent to
type Split =
  Record<”format360p”, URL> |
  Record<”format480p”, URL> |
```

```
  Record<"format720p", URL> |
  Record<"format1080p", URL>

// Equivalent to
type Split =
  { format360p: URL } |
  { format480p: URL } |
  { format720p: URL } |
  { format1080p: URL }
```

Last, but not least, let's build a generic out of it.

```
type Split<Obj> = {
  [Prop in keyof Obj]: Record<Prop, Obj[P]>
}[keyof Obj]

type AvailableFormats = Split<VideoFormatURLs>
```

The moment we change something in `VideoFormatURLs`, we update `AvailableFormats` as well. And TypeScript yells at us with wonderful red squigglies if we have set a property that doesn't exist anymore.

Lesson 33: Mapped Type Modifiers

Our video application allows for signed-in users. Once a user has signed in, they can define preferences on how they want to consume their video content. A simple type modeling user preferences can look like this:

```
type UserPreferences = {
  format: keyof VideoFormatURLs
  subtitles: {
    active: boolean,
    language: keyof SubtitleURLs
  },
  theme: 'dark' | 'light'
}
```

The references to `VideoFormatURLs` and `SubtitleURLs` make sure we don't have to maintain more types than necessary. Updating one of these types adds another part to the union of keys at `format` and `subtitles.language`.

Also, instead of allowing every string to be a valid `theme`, we restrict this property to be either `dark` or `light`.

Partials

As you can read from the type `UserPreferences`, no property is optional. All properties are required to produce a sound user experience, so we don't want to leave anything out. To ensure all keys are set, we provide a set of default user preferences:

```
const defaultUP: UserPreferences = {
  format: 'format1080p',
  subtitles: {
    active: false,
    language: 'english'
  },
  theme: 'light'
}
```

We use a type annotation here. Usually we try to infer as much as possible, but defaults fall into the category of maintained objects. `defaultUP` can change, and the moment we change it we want to validate it against `UserPreferences`.

For our users we just store deltas. If a user changes their preferred video format to something different, it's only the new format that we store.

```
const userPreferences = {
  format: 'format720p'
}
```

To get to the full set of preferences, we merge our default preferences with the user's preferences in a function:

```
function combinePreferences(defaultP, userP)  {
  return { ...defaultP, ...userP }
}
```

Using the object spread syntax, we create an object that is a copy of `defaultP`, and override or extend with all properties from `userP`. The resulting object is the full user preferences, with the delta applied.

Now, let's add types to this function. `defaultP` is easy to type:

```
function combinePreferences(
  defaultP: UserPreferences,
  userP: unknown
) {
  return { ...defaultP, ...userP }
}
```

But how do we type `userP`? We would need a type where every key can be optional, something like this:

```
type OptionalUserPreferences = {
  format?: keyof VideoFormatURLs
  subtitles?: {
```

```
    active?: boolean,
    language?: keyof SubtitleURLs
  },
  theme?: 'dark' | 'light'
}
```

But, of course, we don't want to maintain that type ourselves. Let's create a helper type `Optional` that takes it for us. This is a mapped type, where we modify the property features so each key becomes optional:

```
type Optional<Obj> = {
  [Key in keyof Obj]?: Obj[Key]
}
```

Note the little question mark next to the mapped argument where we iterate through all the keys. This is called a *property modifier*. With that, we create a copy of the type parameter `Obj` where all keys are optional.

Let's annotate our function `combinePreferences` with this helper type.

```
function combinePreferences(
  defaultP: UserPreferences,
  userP: Optional<UserPreferences>
```

```
) {
  return { ...defaultP, ...userP }
}
```

Now, we get extra autocomplete and type safety when using `combinePreferences`.

```
// OK!
const prefs = combinePreferences(
  defaultUP,
  { format: 'format720p' }
)

// boom!
const prefs = combinePreferences(
  defaultUP,
  { format: 'format720p' }
)
```

`Optional<Obj>` is a built-in type in TypeScript called `Partial<Obj>`. It also has a reversed operation `Required<Obj>` which makes all keys required by removing the optional property modifier. It is defined as:

```
type Required<Obj> = {
  [Key in Obj]-?: Obj[Key]
}
```

Readonly

One thing we want to ensure is that `defaultUP` cannot be changed from other parts of our software. It should be maintained in code, not by a side effect. From a tooling perspective, we need a type that ensures every property is a read-only property.

```
type Const<Obj> = {
  readonly [Key in Obj]: Obj[Key]
}
```

You see that we add a property modifier: `readonly`. With that, `defaultUP` won't be updated without TypeScript complaining.

```
const defaultUP: Const<UserPreferences> = {
  format: 'format1080p',
  subtitles: {
    active: false,
    language: 'english'
  },
  theme: 'light'
}
defaultUP.format = 'format720p'
```

`Const<Obj>` is available in TypeScript as `Readonly<Obj>`. In JavaScript we would still be allowed to modify that object.

That's why we use `Object.freeze` to make sure we can't change anything at runtime. The return value's type of `Object.freeze` is `Readonly<Obj>`.

```
function genDefaults(obj: UserPreferences) {
  return Object.freeze(obj)
}

const defaultUP = genDefaults({
  format: 'format1080p',
  subtitles: {
    active: false,
    language: 'english'
  },
  theme: 'light'
})

// defaultUP is Readonly<UserPreferences>
defaultUP.format = 'format720p'
```

This causes an error in both TypeScript *and* JavaScript.

Deep Modifications

There's one thing to keep in mind with `Readonly` and `Partial`: our nested data structure. For example, this call will cause some errors in TypeScript:

```
const prefs = combinePreferences(
  defaultUP,
  { subtitles: { language: 'german' } }
)
```

TypeScript expects us to provide the full object for `subtitles`, as `Partial` just made the first level of properties optional. With the kind of assignment we are doing, this is actually expected behavior. The call above would override our `subtitles` property and delete `subtitles.active`. We would need to create more sophisticated assignments, and also more sophisticated types.

A similar problem pops up when we look at our default preferences. `Readonly` only modifies the first level of properties, which means that this call does not cause an error in TypeScript, whereas it breaks once it runs in the browser:

```
defaultUP.subtitles.language = 'german'
```

To make sure our types are what we expect them to be, we need helper types that go deeper than one level. Thankfully, TypeScript allows for recursive types. We can define a type that references itself, and goes one level deeper. See `DeepReadonly` for instance:

```
type DeepReadonly<Obj> = {
  readonly [Key in Obj]: DeepReadonly<Obj[Key]>
}
```

TypeScript knows to stop the recursion if `Obj[Key]` returns a primitive or value type, or a union of primitive or value types.

Let's apply the new helper type to our `genDefaults` function as return type:

```
function genDefaults(
  obj: UserPreferences
): DeepreadOnly<UserPreferences> {
  return Object.freeze(obj)
}
```

As `Readonly` is a subtype of `DeepReadonly`, the narrower return type of `Object.freeze` is compatible with the wider return type we defined. The same can be done for partials:

```
type DeepPartial<T> = {
  [P in keyof T]?: DeepPartial<T[P]>
}
```

But the details of the new implementation are up to you!

Lesson 34: Binding Generics

Let's revisit `combinePreferences`. We spoke a lot about what arguments we want to pass into this function, that we haven't had a look at what's being returned by our operation.

```
function combinePreferences(
  defaultP: UserPreferences,
  userP: Partial<UserPreferences>
) {
  return { ...defaultP, ...userP }
}
const prefs = combinePreferences(
  defaultUP,
  { format: 'format720p' }
)
```

When we hover over `prefs`, we can see the outcome of what TypeScript infers from our assignment:

```
const prefs: {
  format: "format360p" | "format480p" |
          "format720p" | "format1080p";
  subtitles: {
    active: boolean;
    language: "english" |
              "german" | "french";
  };
  theme: "dark" | "light";
}
```

This is the same as `UserPreferences` and what we expected. With one argument being `UserPreferences`, and the other being `Partial<UserPreferences>`, the combination of both arguments should be the full `UserPreferences` type again.

Getting `UserPreferences` in return from `combinePreferences` is perfectly fine behavior and will make your app a lot more type-safe than it was. Let's take this as an opportunity to explore type annotations, type inference, and generic type binding, and see their effects.

Type Inference

Our user's preferences are a video format of 720p and a dark theme. The corresponding object is:

```
{ format: 'format720p', theme: 'dark' }
```

We use this literal as a literal argument for `combinePreferences`.

```
combinePreferences(
  defaultUP,
  { format: 'format720p', theme: 'dark' }
)
```

The moment we pass the literal, TypeScript infers the type of our literal to be the value type. This is because this value, being an argument of a function, can't change through operations. The only way we can modify this value is by editing the source code. It's final.

When we assign this value to a variable, things are different.

```
const userSettings = {
  format: 'format720p', theme: 'dark'
}

combinePreferences(
  defaultUP, userSettings
)
```

The moment we assign this value to `userSettings`, TypeScript infers its type to the most reasonably widest type. In our case, strings.

```
// typeof userSettings =
{
  format: string,
  theme: string
}
```

This type is much wider than what we expect in our `UserPreferences` type. TypeScript will throw red squigglies to us because we can't take the wider `string` from `"dark" | "light"`, nor for all the formats we listed.

And TypeScript is right! There is no security against changing the value at some point to something entirely incompatible. Thank you, TypeScript!

One thing we could do is add `const` context:

```
const userSettings = {
  format: 'format720p', theme: 'dark'
} as const
```

This protects it from change in TypeScript and narrows the assignment down to its value type, thus being compatible with `Partial<UserPreferences>` as it is a subtype. The other thing we could do is write a type annotation.

Type Annotations

While we advocate for using as much inference as reasonably possible, annotations are a magical thing we can use

when our types are very narrow to not comply with primitive types. Type annotations do a type check the moment we assign a value.

```
const userSettings: Partial<UserPreferences> = {
  format: 'format720p', theme: 'dark'
}
```

With that type annotation, `userSettings` will always be `Partial<UserPreferences>` as long as the values we assign pass the type check. If they do, we will never get back to their original values when using the variable further on. This information is lost to us.

Generic Type Binding

The process of substituting a concrete type for a generic is called *binding*. Let's inspect what happens if we bind a generic type parameter to a concrete type.

This is `combinePreferences` with a generic type parameter.

```
function combinePreferences<
  UserPref extends Partial<UserPreferences>
>(
  defaultP: UserPreferences,
  userP: UserPref
) {
```

```
  return { ...defaultP, ...userP }
}

const prefs = combinePreferences(
  defaultUP,
  { format: 'format720p', theme: 'dark' }
)
```

When we call `combinePreferences` with an annotated type `Partial<UserPreferences>`, we substitute `UserPref` for its supertype. We get the same behavior we had originally.

When we call `combinePreferences` with a literal or a variable in `const` context, we bind the value type to `UserPref`.

1. `{ format: 'format720p', theme: 'dark' }` is taken as literal, therefore we look at the value type.
2. The value type `{ format: 'format720p', theme: 'dark' }` is a subtype of `Partial<UserPreferences>`, so it type-checks.
3. We bind `UserPref` to `{ format: 'format720p', theme: 'dark' }`, which means we now work with the value type, instead of `Partial<UserPreferences>`.

`UserPref` has changed now, which means that our result type has changed as well. If we hover over `prefs`, we get the following type information:

```
const p: {
  format: “format360p” | “format480p” |
          “format720p” | “format1080p”;
  subtitles: {
    active: boolean;
    language: “english” |
              “german” | “french”;
  };
  theme: “light” | “dark”;
} & {
  format: “format720p”;
  theme: “dark”;
}
```

First, we learn what the operation `{...defaultP, ...userP}` actually does. It creates a combination of two objects, and the resulting type is an intersection. This makes sense!

We also see what `UserPrefs` became the moment we passed a literal: the value type of said literal.

This intersection creates an interesting behavior. We have a couple of union types that are now intersected with subtypes of their sets. In such a scenario, the narrower set always wins:

```
('dark' | 'light') & 'dark' // type is 'dark'
```

Which means we know exactly which values we get when we work with `prefs`:

```
prefs.theme // is of type 'dark'
prefs.format // is of type 'format720p'
```

This makes some checks in our code easier. Be careful, though, with too many value types. If we take the same pattern for the default preferences and pass a `const` context object to it, we might get some unwanted side effects:

```
function combinePreferences<
  Defaults extends UserPreferences,
  UserPref extends Partial<UserPreferences>
>(
  defaultP: Defaults,
  userP: UserPref
) {
  return { ...defaultP, ...userP }
}

const defaultUP = {
  //  wW know what we have here
} as const

const prefs = combinePreferences(
  defaultUP,
  { format: 'format720p', theme: 'dark' }
)
```

The resulting type looks like this:

```
const prefs: {
  readonly format: “format1080p”;
  readonly subtitles: {
    readonly active: false;
    readonly language: “english”;
  };
  readonly theme: “light”;
} & {
  format: “format720p”;
  theme: “dark”;
}
```

The intersection of two distinct value types always results in `never`, which means that both `theme` and `format` become unusable to us.

Lesson 35: Generic Type Defaults

In the last lesson of this chapter we want to show videos inside a `video` element. To make it easier for ourselves and our co-workers, we decide to abstract handling with the DOM, and we choose to use classes for this.

The class should behave like this:

1. We can instantiate as many as we like and pass our user preferences to it. The user preferences are important to select the right video format URL.
2. We can attach any HTML element to it. If it's a `video` element, we load the video source directly. If it's any other element, we use it as a wrapper for a newly created `video` element. `video` elements are the default, though.
3. The element is not required for instantiation; we can set it at a later stage. This means the element can be `undefined` at the moment we load a video.

Let's implement this class.

Moving to Generics

First, we create a helper type `Nullable` that adds `undefined` in a union. This makes reading field types of classes much easier.

```
type Nullable<G> = G | undefined
```

Next, we start with the class. We set the type of the element to `HTMLElement` as this is the supertype of all HTML elements.

```
class Container {
```

```
  #element: Nullable<HTMLElement>;
  #prefs: UserPreferences

  // We only require the user preferences
  // to be set at instantiation
  constructor(prefs: UserPreferences) {
    this.#prefs = prefs
  }

  // We can set the element to an HTML element
  set element(value: Nullable<HTMLElement>) {
    this.#element = value
  }

  get element(): Nullable<HTMLElement>  {
    return this.#element
  }

  // We load the video inside a video element.
  // If #element isn't an HTMLVideoElement, we
  // create one and append it to #element
  loadVideo(formats: VideoFormatURLs) {
    const selectedFormat =
      formats[this.#prefs.format].href
    if(this.#element instanceof HTMLVideoElement) {
      this.#element.src = selectedFormat
    } else if(this.#element) {
      const vid = document.createElement('video')
      this.#element.appendChild(vid)
      vid.src = selectedFormat
    }
  }
}
```

And this already works wonderfully:

```
const container = new Container(userPrefs)
container.element = document.createElement('video')
container.loadVideo(videos)
```

`HTMLElement` can be way too generic for some tastes. Especially when we deal with videos, we might want to work with the video functions of `HTMLVideoElement`. And when working with that, we need the right type information.

Generics can help. We can pinpoint the exact type we are dealing with, and with type constraints we can make sure it's an extension of our supertype `HTMLElement`.

```
class Container<GElement extends HTMLElement> {
  #element: Nullable<GElement>;

  // ...abridged...

  set element(value: Nullable<GElement>) {
    this.#element = value
  }
  get element(): Nullable<GElement>  {
    return this.#element
  }

  // ...abridged...
}
```

This is better, but we are not entirely happy with it yet.

Adding Defaults

As we lack a concrete element in the constructor, TypeScript has nothing to infer to bind `GElement` to a concrete type. We fall back to the supertype, `HTMLElement` without an explicit generic annotation:

```
// container accepts any HTML element
const container
  = new Container(userPrefs)

// container accepts HTMLVideoElement
const vidcontainer
  = new Container<HTMLVideoElement>(userPrefs)
```

And this is bad, as our default should always be `HTMLVideoElement`. Other elements are the exception. This is where generic default parameters come in. If we don't provide a generic annotation, TypeScript will use the default parameter as type.

```
class Container<
  GElement extends HTMLElement = HTMLVideoElement> {
  // ...
```

```
}

// container accepts HTMLVideoElement
const container = new Container(userPrefs)
```

Compared to type constraints, generic default parameters don't create a boundary, but a default value in case we can't infer or don't annotate. If a generic default parameter exists without a boundary, the generic can accept `any`. Like function default parameters, generic default parameters have to come last in a generic definition.

Generic default parameters are extremely useful for classes that need to bind a generic but don't have the information at instantiation. For all other cases, type constraints work best.

Generic Default Parameters and Type Inference

While generic default parameters can be extremely powerful, we also have to be very cautious. Take this function that does something similar to the `Container` class. It loads a video in an element, and differentiates between the following cases:

1. If we don't provide an element, we create a `video` element
2. If we provide a `video` element, we load the video in this element
3. If we provide any other element, we use this as a wrapper for a new `video` element.

The function returns the element we passed as an argument for further operations. With generic default parameters we can beautifully define this behavior, and rely only on type inference:

```
declare function createVid<
  GElement extends HTMLElement = HTMLVideoElement
>(
  prefs: UserPreferences,
  formats: VideoFormatURLs,
  element?: GElement
)
```

If we try it out, we get the following:

```
declare const userPrefs: UserPreferences
declare const formats: VideoFormatURLs

// a is HTMLVideoElement, the default!
const a = createVid(userPrefs, formats)
```

```
// b is HTMLDivElement
const b = createVid(
    userPrefs, formats,
    document.createElement('div'))

// c is HTMLVideoElement
const c = createVid(
    userPrefs, formats,
    document.createElement('video'))
```

However, this only works when we rely solely on type inference. Generics also allow us to bind the type explicitly.

```
const a
  = createVid<HTMLAudioElement>(userPrefs, formats)
```

`a` is of type `HTMLAudioElement`, even though our implementation will return an `HTMLVideoElement`. Also, since we are on the type level, the implementation has no clue that we want to have an `HTMLAudioElement`. That's why we need to be cautious when we use generic default parameters. Also, we have a much better tool for cases like that, as we will see in the next chapter.

Recap

Generics allow us to prepare for types we don't know up front. This allows us to design robust APIs with better type information, and make sure that we only pass values where our types match certain criteria.

1. With generics we made sure that we don't have to create more functions just to please the type system. Generics allow us to generalize functions for broader usage.
2. Generic constraints allow us to create boundaries. Instead of accepting *anything* for our generic types, we are allowed to set some criteria, such as the existence of certain keys or types of properties.
3. Generics also allow us to work better with object keys. Depending on what we pass as an argument to a function, we can infer the right keys and let TypeScript throw red squigglies at us if we don't provide the correct arguments.
4. Generics work extraordinarily well with mapped types. Through maps of union keys, index access types, and the `Record` helper, we are able to create a type that allows us to split an object type into a set of unions.
5. Mapped type modifiers allow us to copy an object type, but set all properties as optional, required, or read-only.

6. We learned a lot about binding generics. The moment we substitute a generic type for a real one is crucial to understand the TypeScript type system.
7. We also saw how generic classes work, and how we use generic type defaults to make our life a little bit easier.

Working with generics is key to getting the most out of TypeScript's type system. Generics were designed to conform to the majority of real-world JavaScript scenarios, and open doors to even better and more robust type information. The next chapter will take us even further!

Interlude: On Names

Generics are not something entirely new to programming. They have been around for a long time: one of the first programming languages, Ada, introduced a generic concept in the late 1970s.

Syntax-wise, generics as we use them today in TypeScript are a descendant of C++ templates. This comes as no surprise, as Java, JavaScript, C#, and many other languages are heavily inspired by the way C/C++ described its programs. For generics, TypeScript borrows the angle brackets syntax.

Even though C++ templates are much more powerful than type substitution, the syntax has led to naming generic type parameters mostly `T`, for template. Subsequent parameters usually go either along the alphabet (`U`, `V`, `W`) or are `P` for property, `K` for key, and so on.

This can result in highly unreadable types. If I give `Record<T, U>` to you, with no understanding of what `Record` does, you might wonder what we should expect from the types we pass along. A `Record<Obj, PropType>` might be clearer: we can pass objects and types for properties.

So even though it is common to use single letter generics, I advise you to do better. Types should be documentation, and

that's why we have to be as explicit as possible with generic type parameters. This is my style guide:

1. Uppercase words, no single letters. Uppercase to differentiate it from function parameters.
2. Highly abbreviated, but still readable. `Obj` is clearer than `O`, shorter than `Object`. `URLObj` indicates it is an object with URL properties.
3. Use prefixes to differentiate from actual types. For example, the type `Element` exists in the DOM API. I use `GElement` for my generic type parameter (`Elem` is also an option).
4. `G` is a prefix for `generic`, but we can be clearer if we handle keys. `URLObject` is an object with URLs, so `UKey` is a key from this object.

It's not a lot, but it makes generics a lot more readable.

Chapter Six

CONDITIONAL TYPES

Conditional Types

In chapter 4 we learned how to move through the type space with union and intersection types, and how to create specific sets of values for our data structures. In chapter 5 we generalized type behavior and bound types at a later stage, making our functions and classes flexible yet specific to defined sets.

But what if the behavior of our types is ambiguous? What if there's more than one answer to a generic type? Or the type output simply "depends"? Y'know – how it is in JavaScript all the time.

With conditional types, we get the last tool in our tool belt to make most sense out of JavaScript code. Conditional types allow us to validate an input type's set, and decide on an output type based on this condition: `if-else` statements, but on a type level.

This sounds complicated. To be sure, some conditional types can be mind-blowingly hard to understand, and their potential is sometimes hard to grasp. But this is what we want to clear up! Let's make type-level arithmetic approachable and usable!

To illustrate the features of conditional types, we are going to look at an e-commerce application that sells physical audio media to collectors: CDs, vinyl LPs, and even cassette tapes!

Lesson 36: If This, Then That

Consider the following data structure we set up for our e-commerce shop. We have customers, products, and orders. Customers have an ID, a first name, and a last name.

```
type Customer = {
  customerId: number,
  firstName: string,
  lastName: string
}

const customer = {
  id: 1,
  firstName: 'Stefan',
  lastName: 'Baumgartner'
} // = type Customer
```

The product has a product ID, a title, and a price.

```
type Product = {
  productId: number,
  title: string,
  price: number
}

const product = {
  id: 22,
  title: 'Form Design Patterns',
  price: 29
}
```

The order has an ID as well, a customer (of type `Customer`), a list of products within the order, and a date.

```
type Order = {
  orderId: number,
  customer: Customer,
  products: Product[],
  date: Date
}
```

This is a much simplified but robust start for our little app. We are implementing an administration interface for an e-commerce application.

We want to provide a `fetchOrder` function, which works as follows:

1. If we pass a customer, we get a list of orders from this customer.
2. If we pass a product, we get a list of orders that include this product.
3. If we pass an order ID, we just get that particular order.

Our first idea to implement this would be function overloads.

```
function fetchOrder(customer: Customer): Order[]
function fetchOrder(product: Product): Order[]
```

```
function fetchOrder(orderId: number): Order
function fetchOrder(param: any): any {
  // Implementation to follow
}
```

This works well for simple cases where we're absolutely sure which parameters we expect:

```
fetchOrder(customer) // It's Order[]
fetchOrder(2) // It's Order
```

But it gets hairy when our input is ambiguous. When we pass an argument that can be either `Customer` or `number`, the output is a bit boring:

```
declare const ambiguous: Customer | number

fetchOrder(ambiguous) // It's any
```

Of course, we could patch the types of the implementation function to be a bit clearer:

```
function fetchOrder(customer: Customer): Order[]
function fetchOrder(product: Product): Order[]
function fetchOrder(orderId: number): Order
```

```
function fetchOrder(
  param: Customer | Product | number
): Order[] | Order {
  // Implementation to follow
}
```

But being explicit about all possible outcomes gets very verbose very quickly:

```
function fetchOrder(customer: Customer): Order[]
function fetchOrder(product: Product): Order[]
function fetchOrder(orderId: number): Order
function fetchOrder(
  param: Customer | Product
): Order[]
function fetchOrder(
  param: Customer | number
): Order[] | Order
function fetchOrder(
  param: Product | number
): Order[] | Order
function fetchOrder(
  param: Customer | Product | number
): Order[] | Order {
  // I hope I didn't forget anything
}
```

Seven overloads for three possible input types, and two possible output types. Now add another one, it's exhausting!

Enter Conditional Types

This has to be easier. We can map each input type to an output type

- If the input type is `Customer`, the return type is `Order[]`
- If the input type is `Product`, the return type is `Order[]`
- If the input type is `number`, the return type is `Order`

If the input type is a combination of available input types, the return types are a combination of the respective output types.

We can model this behavior with conditional types. The syntax for conditional types is based on generics and is as follows:

```
type Conditional<T> = T extends U ? A : B
```

Where `T` is the generic type parameter. `U`, `A`, and `B` are other types. We can read this statement like ternary operations in JavaScript:

```
const x = (t > 0.5) ? true : false
```

The statement above reads that if `t` is bigger than `0.5`, then `x` is `true`, otherwise it's `false`. We can read the conditional

type statement in the same way: if type `T` extends type `U`, the assigned type is `A`, otherwise it's `B`.

Let's see how this works with the `fetchOrder` function. First, we create a type for all possible inputs.

```
type FetchParams = number
  | Customer
  | Product;
```

Then, we create a generic type `FetchReturn<T>` with a generic constraint to `FetchParams`.

```
type FetchReturn<Param extends FetchParams> =
  Param extends Customer ? Order[] :
  Param extends Product ? Order[] : Order
```

The type constraint `<Param extends FetchParams>` already reduces the available input types to three possible types, so this condition is already checked. The conditional then reads:

1. If the `Param` type extends `Customer`, we expect an `Order[]` array.
2. Else, if `Param` extends `Product`, we also expect an `Order[]` array.

3. Otherwise, when only `number` is left, we expect a single `Order`.

In TypeScript jargon, we say the conditional type *resolves* to `Order[]`.

Let's adapt our function to work with the new conditional type:

```
function fetchOrder<Param extends FetchParams>(
  param: Param
): FetchReturn<Param> {
  // Well, the implementation
}
```

This is all we need to get the required return types for every combination of input types.

```
fetchOrder(customer) // Order[] OK!
fetchOrder(product) // Order[] OK!
fetchOrder(2) // Order[] OK!

fetchOrder(ambiguous) // Order | Order[]

declare x: any

// any is not part of `FetchParams`
fetchOrder(x)
```

Conditional types also work well with the idea of having a type layer around regular JavaScript. They work only in the type layer and can be easily erased, while still being able to describe all possible outcomes of a function.

Lesson 37: Combining Function Overloads and Conditional Types

In the previous lesson we stated that conditional types are capable of describing everything that function overloads can do, and are much more correct. While this is technically true, there are scenarios where a healthy mix of function overloads and conditional types create much better readability and clearer outcomes.

One such scenario is dealing with optional arguments. The `fetchOrder` function is synchronous. And as we know, fetching something from a database or a back end most of the time happens asynchronously.

Let's refactor `fetchOrder` so it allows for asynchronous data retrieval. The function should combine two different asynchronous patterns:

1. If we pass a single argument (either `number`,

`Customer` or `Product`), we get a promise in return with the respective outcome (`Order` or `Order[]`).

2. We are able to pass a callback as a second argument. This callback gets the result (`Order` or `Order[]`) as a parameter; the function `fetchOrder` returns void.

This is a classical pattern that we can see in many Node.js libraries. Either we pass a callback, or we return a promise. The interesting part of this example is that the second argument is entirely optional. This means that the function shape can be very different. Let's look at each function head separately.

```
// A callback helper type
type Callback<Res> = (result: Res) => void

// Version 1. Similar to the version from
// the previous lesson, but wrapped in a promise
function fetchOrder<Par extends FetchParams>(
  inp: Par
): Promise<FetchReturn<Par>>

// Version 2. We pass a callback function that
// gets the result, and return void.
function fetchOrder<Par extends FetchParams>(
  inp: Par, fun: Callback<FetchReturn<Par>>
): void
```

With a function shape that is so different, it's not sufficient enough to do a conditional type for a simple union.

Tuple Types for Function Heads

A possible solution would be to do a conditional type for a union of the entire set of function heads.

In JavaScript, we have the possibility to condense all function arguments into a tuple with rest parameters.

```
function doSomething(...rest) {
  return rest[0] + rest[1]
}

// Returns "JavaScript"
doSomething('Java', 'Script')
```

This rest parameter can be typed as a tuple. Let's type the callback version's arguments as a tuple:

```
function fetchOrder<Par extends FetchParams>(
  ...args: [Par, Callback<FetchReturn<Par>>]
): void
```

And, let's also type the promise version's arguments as a tuple.

```
function fetchOrder<Par extends FetchParams>(
  ...args: [Par]
): Promise<FetchReturn<Par>>
```

We sum up the entire argument list of each function head into separate tuple types. This means that we can create a conditional type that selects the right output type.

```
// A small helper type to make it easier to
// read
type FetchCb<T extends FetchParams> =
  Callback<FetchReturn<T>>

type AsyncResult<
  FHead, Par extends FetchParams
> = FHead extends [Par, FetchCb<Par>>] ? void :
    FHead extends [Par] ? Promise<FetchReturn<T>> :
    never;
```

The conditional type reads as follows:

1. If the function head `FHead` is a subtype of tuple `FetchParams` and `FetchCb`, then return void.
2. Otherwise, if the function head is a subtype of the tuple `FetchParams`, return a promise.

3. Otherwise, never return

We can use this newly created conditional type and bind it to our function.

```
function fetchOrder<
  Par extends FetchParam,
  FHead
>(...args: FHead) : AsyncResult<FHead, Par>
```

And this pretty much does the trick. But it also comes at a high price:

1. **Readability.** Conditional types are already hard to read. In this case, we have two nested conditional types: the old `FetchReturn`, that reliably returns the respective return type; and the new `AsyncResult`, that tells us if we get `void` or a promise back.
2. **Correctness.** Somewhere along the way we might lose binding information for our generic type parameters. This means we don't get the *actual* return type, but a union of all possible return types. Making sure we don't lose anything requires us to bind a lot of parameters, thus crowding our generic signatures and generic constraints.

In cases like this, it might be a better idea to still rely on function overloads.

Function Overloads Are Fine

Rewind to our initial function description. We described two possible versions and their outcomes. This mirrors exactly the function overloads we would've done without conditional types. So let's see how we can implement the whole set of possible functions:

```
// Version 1
function fetchOrder<Par extends FetchParams>(
  inp: Par
): Promise<FetchReturn<Par>>
// Version 2
function fetchOrder<Par extends FetchParams>(
  inp: Par, fun: Callback<FetchReturn<Par>>
): void
// The implementation!
function fetchOrder<Par extends FetchParams>(
  inp: Par, fun?: Callback<FetchReturn<Par>>
): Promise<FetchReturn<Par>> | void {
  // Fetch the result
  const res =
    fetch(`/backend?inp=${JSON.stringify(inp)}`)
    .then(res => res.json())

  // If there's a callback, call it
  if(fun) {
```

```
    res.then(result => {
      fun(result)
    })
  } else {
    // Otherwise return the result promise
    return res
  }
}
```

If we look closely, we see that we don't leave conditional types completely. The way we treat the `FetchReturn` type is still a conditional type, based on the `FetchParams` union type. The variety of inputs and outputs was nicely condensed into a single type.

However, the complexity of different function heads was better suited to function overloads. The input and output behavior is clear and easy to understand, and the function shape is different enough to qualify for being defined explicitly. As a rule of thumb for your functions:

1. If your input arguments rely on union types, and you need to select a respective return type, then a conditional type is the way to go.
2. If the function shape is different (e.g. optional arguments), and the relationship between input arguments

and output types is easy to follow, a function overload will do the trick.

Lesson 38: Distributive Conditionals

Before we continue into the realms of conditional types with more examples, let's hang out with the one conditional type we just wrote.

```
type FetchParams = number
  | Customer
  | Product

type FetchReturn<Param extends FetchParams> =
  Param extends Customer ? Order[] :
  Param extends Product ? Order[] : Order
```

Remember the metaphor from the previous chapter: generics work like functions, have parameters, and return output. With that in mind, we can see how this conditional type works when we put in one type as argument.

Let's bind `Param` to `Customer`:

```
type FetchByCustomer = FetchReturn<Customer>
```

Substitute with the conditional's definition:

```
type FetchByCustomer =
  Customer extends Customer ? Order[] :
  Customer extends Product ? Order[] : Order
```

Run through the conditions and get to a result.

```
type FetchByCustomer = Order[]
```

We can run through the same process with all other compatible types.

Distribution over Unions

It gets a little different once we pass union types. In most cases, conditional types are distributed over unions during instantiation. Let's see how this works in practice.

First, we instantiate `FetchParam` with a union of `Product` and `number`.

```
type FetchByProductOrId =
  FetchReturn<Product | number>
```

`FetchReturn` is a distributive conditional type. This means that each constituent of the generic type parameter is instantiated with the same conditional type. In short: a conditional type of a union type is like a union of conditional types.

```
type FetchByProductOrId =
  (
    Product extends Customer ? Order[] :
    Product extends Product ? Order[] : Order
  ) |
  (
    number extends Customer ? Order[] :
    number extends Product ? Order[] : Order
  )
```

Again, we run through the conditions to get a result.

```
type FetchByProductOrId = Order[] | Order
```

And this is our expected result!

Knowing that TypeScript's conditional types work through distribution is incredibly important for a couple of reasons.

1. We can track each input type to exactly one output type, no matter in which combination they occur.

2. This means that in a scenario like ours, where we want to have different return types for different input types, we can be sure we don't forget a combination. The possible combinations of return types is exactly the possible combinations of input types.

Even though the possible combinations are the same, return type unions remove duplicates and impossible results. This means that if we do a distribution over all possible input types, we get two output types in the result:

```
type FetchByProductOrId =
  FetchReturn<Product | Customer | number>

// Equal to

type FetchByProductOrId =
  (
    Product extends Customer ? Order[] :
    Product extends Product ? Order[] : Order
  ) |
  (
    Customer extends Customer ? Order[] :
    Customer extends Product ? Order[] : Order
  ) |
  (
    number extends Customer ? Order[] :
    number extends Product ? Order[] : Order
  )
```

```
// Equal to

type FetchByProductOrId =
  Order[] | Order[] | Order

// Removed redundancies

type FetchByProductOrId =
  Order[] | Order
```

This is a feature that will be important later in this chapter.

Naked Types

An important precondition to distributive conditional types is that the generic type parameter is a *naked* type. Naked type is type system jargon and means that the type parameter is present *as is*, without being part of any other construct.

Being naked is the most common case for generic type parameters. The non-naked version can lead to interesting side effects. Let's wrap the type parameter in a tuple type.

```
type FetchReturn<Param extends FetchParams> =
  [Param] extends [Customer] ? Order[] :
  [Param] extends [Product]  ? Order[] : Order
```

For single type bindings, the conditional type works as before:

```
type FetchByCustomer = FetchReturn<Customer>

type FetchByCustomer =
  // This condition is still true!
  [Customer] extends [Customer] ? Order[] :
  [Customer] extends [Product] ? Order[] : Order

type FetchByCustomer = Order[]
```

The tuple `[Param]` when instantiated with `Customer` is still a subtype of the tuple `[Customer]`, so this condition still resolves to `Order[]`. When we instantiate `Param` with a union type, and this doesn't get distributed, we get the following result:

```
type FetchByCustomerOrId
  = FetchReturn<Customer | number >

type FetchByProductOrId =
  // This is false!
  [Customer | number] extends [Customer] ? Order[] :
  // This is obviously also false
  [Customer | number] extends [Product] ? Order[] :
  // So we resolve to this
  Order

type FetchByProductOrId = Order // Gasp!
```

`[Customer | number]`, being a wider type, is a supertype of `[Customer]` and therefore doesn't extend `[Customer]`. No condition applies, and our conditional type falls through to the last option, `Order`. And this is a false result.

To make this conditional type a lot safer and more correct, we can add another condition to it where we check for the subtype of `number`. The last conditional branch resolves to `never`.

```
type FetchReturn<Param extends FetchParams> =
  [Param] extends [Customer] ? Order[] :
  [Param] extends [Product]  ? Order[] :
  [Param] extends [number] ? Order : never
```

This ensures we definitely get the correct return value if we work with a single type. Union types always resolve to `never`, which can be a nice way of making sure that we first narrow down to a single constituent of the union.

Lesson 39: Filtering with never

The distributive property of conditional types allows for some interesting use cases when combined with `never`. It's possible to create useful filter types as building blocks for advanced, self-maintaining types in our applications. Remember the ultimate goal: we want to model data and behavior but never maintain our types beyond the model.

The Model

Our e-commerce application gets another feature. We want to create CDs and LPs with a `createMedium` function. This is how our model looks.

The type `Medium` contains our base properties:

```
type Medium = {
  id: number,
  title: string,
  artist: string,
}
```

`TrackInfo` stores the number of tracks and the total duration.

```
type TrackInfo = {
  duration: number,
  tracks: number
}
```

A `CD` is a combination of `Medium` and `TrackInfo`. We also add a `kind` to create discriminated unions.

```
type CD = Medium & TrackInfo & {
  kind: 'cd'
}
```

An `LP` is also derived from the base `Medium` class. It contains two sides which each store `TrackInfo`:

```
type LP = Medium & {
  sides: {
    a: TrackInfo,
    b: TrackInfo
  },
  kind: 'lp'
}
```

We combine all possible media in an `AllMedia` union type. We also define a union of media keys.

```
type AllMedia = CD | LP
type MediaKinds = AllMedia['kind'] // 'lp' | 'cd'
```

These are our types. The function `createMedium` should work as follows:

1. The first argument is the type we want to create, either an LP or a CD.
2. The second argument is all the missing info we need to successfully create this medium. We don't need the properties type, which we defined in our first argument, nor the ID, as this will be generated by the function.

3. The function returns the newly created medium.

On to the implementation.

Select Branches of Unions

The bare minimum of the function head defines two types for the arguments and `AllMedia` for the return type. Alternatively, this can be `Medium` to point to the base type.

```
declare function createMedium(kind, info): AllMedia
```

The first type we can define is which kind we want to select. It has to be of type `MediaKinds`.

```
declare function createMedium(
  kind: MediaKinds, info
): AllMedia
```

We use a generic to bind the actual value type if we use a literal.

```
declare function createMedium<
  Kin extends MediaKinds
```

```
>(
  kind: Kin, info
): AllMedia
```

Now that we know which kind of medium we want to create, we can focus on the expected output. `AllMedia` is definitely too wide, but how can we select a certain branch in our union?

Remember that conditional types are distributed over union types, meaning that a conditional of unions is like a union of conditionals. We can use this behavior to create a conditional type that checks if each constituent of a union is a subtype of the kind we are filtering for. If so, we return the constituent. If not, we return `never`.

```
type SelectBranch<Brnch, Kin> =
  Brnch extends { kind: Kin } ? Brnch : never
```

Note the naked type `Brnch`! Let's see what happens if we run `AllMedia` through it, and select the branch for `cd`.

```
// We create a type where we want to select the
// cd branch of the AllMedia Union
type SelectCD = SelectBranch<AllMedia, 'cd'>
```

```
// This equals
type SelectCD = SelectBranch<CD | LP, 'cd'>

// A conditional of unions is like a union of
// conditionals
type SelectCD =
  SelectBranch<CD, 'cd'> |
  SelectBranch<LP, 'cd'>

// Substitute for the implementation
type SelectCD =
  (CD extends { kind: 'cd' } ? CD : never) |
  (LP extends { kind: 'cd' } ? LP : never)

// Evaluate!
type SelectCD =
  // This is true! Awesome! Let's return CD
  (CD extends { kind: 'cd' } ? CD : never) |
  // This is false. let's return never
  (LP extends { kind: 'cd' } ? LP : never)

// Equal to
type SelectCD = CD | never
```

We end with a union of `CD | never`. Again, for each constituent of a union we get a proper type. However, `never` is the impossible type. As it says in the name, this can never happen! That's why TypeScript removes everything that resolves to `never` from the union, if there are other constituents available. So `SelectBranch<AllMedia, 'cd'>` resolves to `CD` eventually.

Let's update our return type.

```
declare function createMedium<
  Kin extends MediaKinds
>(
  kind: Kin, info
): SelectBranch<AllMedia, Kin>
```

By binding `Kin` to the value type, we get the correct branch of our union type. Handy! Also, if you use the pattern of adding a `kind` property to create discriminated unions a lot, the `SelectBranch` type becomes a reusable helper type in your arsenal of types.

Extract

A much more generic type is the built-in helper type `Extract<A, B>`. `Extract<A, B>` is defined as.

```
type Extract<A, B> = A extends B ? A : never
```

The documentation says that it extracts from `A` those types that are assignable to `B`. This can be a set of keys, or (in our case) objects.

```
// Resolves to LP
type SelectLP = Extract<AllMedia, { kind: 'lp' }>
```

The moment we add another medium to the union type `AllMedia`, all our types are updated automatically. We have new kinds we can pass to `createMedium`, but also know that we're getting another medium back. No maintenance from our side. We just add something to the model.

Lesson 40: Composing Helper Types

A quick look back to the previous lesson. This is how far we got:

```
declare function createMedium<
  Kin extends MediaKinds
>(
  kind: Kin, info
): SelectBranch<AllMedia, Kin>
```

We select a certain kind, and know which return type to expect.

```
createMedium('lp', { /* tbd */ }) // Returns LP!
createMedium('cd', { /* tbd */ }) // Returns CD!
```

Now we want to focus on the missing information. Remember, we want to add everything that's necessary to create a full medium, except for `id`, which is autogenerated by `createMedium`, or `kind`, which we already defined.

Exclude

This means we need to pass objects that look like this:

```
type CDInfo = {
  title: string,
  description: string,
  tracks: number,
  duration: number
}

type LPInfo = {
  title: string,
  description: string,
  sides: {
    a: {
      tracks: number,
      duration: number
    },
    b: {
      tracks: number,
      duration: number
    }
  }
}
```

But we don't want those types to be maintained – we want to have them autogenerated.

The first thing we want to take care of is knowing which keys of our object we actually need. The best way to do this is by knowing which keys we *don't* need: `kind` and `id`.

```
type Removable = 'kind' | 'id'
```

Good. Now we need to filter all property keys that are not in this set of keys. For that, we create another distributive conditional type. It looks very similar to `Extract`, but resolves differently.

```
type Remove<A, B> = A extends B ? never : A
```

It reads that if the type `A` is part of `B`, remove it (`never`); otherwise keep it. Let's see what happens if we use all keys of `CD` and distribute the union over the `Remove` type. Remember, a conditional of a union is like a union of conditionals.

```
// First our keys
type CDKeys = keyof CD
// Equal to
type CDKeys = 'id' | 'description' |
  'title' | 'kind' | 'tracks' | 'duration'
```

```
// Now for the keys we actually want
type CDInfoKeys = Remove<CDKeys, Removable>

// Equal to
type CDInfoKeys =
  Remove<'id' | 'description' | 'title' |
  'kind' | 'tracks' | 'duration', 'id' | 'kind'>

// A conditional of a union
// is a union of conditionals
type CDInfoKeys =
  Remove<'id', 'id' | 'kind'> |
  Remove<'description', 'id' | 'kind'> |
  Remove<'title', 'id' | 'kind'> |
  Remove<'kind', 'id' | 'kind'> |
  Remove<'tracks', 'id' | 'kind'> |
  Remove<'duration', 'id' | 'kind'>

// Substitute
type CDInfoKeys =
  ('id' extends 'id' | 'kind' ?
    never : 'id') |
  ('description' extends 'id' | 'kind'
    ? never : 'description') |
  ('title' extends 'id' | 'kind'
    ? never : 'title') |
  ('kind' extends 'id' | 'kind' ? never : 'kind') |
  ('tracks' extends 'id' | 'kind' ? never : 'tracks') |
  ('duration' extends 'id' | 'kind' ? never :
'duration')

// Evaluate
type CDInfoKeys =
```

```
  never  |  'description' | 'title' | never |
  'tracks' | 'duration'

// Remove impossible types from the union
type CDInfoKeys =
  'description' | 'title' | 'tracks' | 'duration'
```

Wow, what a process! But we get one step closer to the result we expect. The `Remove` type is built-in to TypeScript and called `Exclude`. The definition is exactly the same, and it's description says that it excludes types from `A` which are in `B`. This is what just happened.

Omit

We now have to take this new set of keys – a subset of the original set of keys – and create an object type with the new keys, which need to be of the type of the original object.

This sounds a lot like a mapped type, doesn't it? Remember `Pick`? `Pick` runs over a set of keys and selects the type from the original property type. This is exactly what we're looking for.

```
type CDInfo = Pick<
  CD,
  Exclude<keyof CD, 'kind' | 'id'>
>
```

How do we read this new type? We *pick* from *CD* all *keys of CD*, but exclude *kind* and *id*. The result is the type we originally envisioned. Once again, generic types behave like functions. They have parameters and an output, and are composable.

Reading this type might feel like a little tongue twister. That's why TypeScript has a built-in type for exactly this combination of `Pick` and `Exclude`, called `Omit`.

```
type CDInfo = Omit<CD, 'kind' | 'id'>
```

We've come a long way with our types. The last step is to compose everything in our `createMedium` function. To successfully omit `kind` and `id` from our medium types, we need to pass the selected branch to `Omit`. Another helper type makes this a bit more readable.

```
type RemovableKeys = 'kind' | 'id'
type GetInfo<Med> = Omit<Med, RemovableKeys>

declare function createMedium<
  Kin extends MediaKinds
>(
  kind: Kin,
  info: GetInfo<SelectedBranch<AllMedia, Kin>>
): SelectBranch<AllMedia, Kin>
```

And that's it! Now TypeScript prompts us only for the properties that are missing. We don't have to specify redundant information, and we get autocomplete and type safety when using our `createMedium` function.

A Set of Helper Types

Being able to compose generics and distributive conditional types allows for a set of smaller, single-purpose helper types that can be assembled for different scenarios. This allows us to define type behavior without maintaining too many types. Focus on the model, describe behavior with helpers.

Lesson 41: The infer Keyword

When working efficiently with TypeScript, we want to keep type maintenance as low as possible. Types should help us be productive, after all, and not stand in the way of what we're trying to achieve.

Up until now we've stuck to a clear workflow: model data, describe behavior. We don't want too much time spent on type maintenance if we can create types dynamically from other types. There are times, however, when we are not sure how our model will look. Especially during development,

things can change. Data can be added and removed, and the overall shape of an object is in flux. This is OK. It's the flexibility JavaScript is known for!

Think of extending our e-commerce admin application with a function that creates users who are allowed to read and modify orders, products, and customers. The function might look something like this:

```
// A userId variable counting up... not safe
// but we are in development mode
let userId = 0

function createUser(name, roles) {
  return {
    userId: userId++,
    name,
    roles,
    createdAt: new Date()
  }
}
```

Pretty straightforward: two generated properties, and the others are just added to the object. Notice the absence of types! Later on, our function might get more concrete. The roles are divided between

- admin: allowed to read and modify everything.
- maintenance: allowed to modify products.

- shipping: allowed to read orders to get the necessary info to dispatch them.

```
function createUser(
  name: string,
  role: 'admin' | 'maintenance' | 'shipping',
  isActive: boolean
) {
  return {
    userId: userId++,
    name,
    role,
    isActive,
    createdAt: new Date()
  }
}
```

But this is just another mutation. Input types get more concrete, and the return type adapts to the changes. We would always have to maintain a type `User` if we want to continue to work with users in a type-safe environment.

Infer the Return Type

It would help a lot if we could *name* the type that gets returned by `createUser`. TypeScript can infer types through assignments. The variable's type takes on the shape of what's returned by the function.

```
// The type of user is the shape returned by
createuser
const user = createUser('Stefan', 'shipping', true)
```

We can put a name on this with the `typeof` operator:

```
/*
type User = {
  userId: number,
  name: string,
  role: 'admin' | 'maintenace' | 'shipping',
  isActive: boolean,
  createdAt: Date
}
*/
type User = typeof user
```

This gets us the `User` type but at a very high price. We always have to call the function to obtain the shape of the return type. What if this function call performs a critical data mutation, in a database, for example? Are database transactions justifiable only to retrieve the type of an object?

What we want is to retrieve the return type from the function signature. For situations like that, TypeScript allows us to infer type variables in the `extends` clause of a conditional type.

Let's create a `GetReturn` type that takes a function. Any function. For now, we want to check if the passed type is a valid function.

```
type GetReturn<Fun> =
  Fun extends (...args: any[]) => any ? Fun : never
```

We combine all possible arguments into an argument tuple (see rest parameters in lesson 37). We know that we have `any` return type. If we pass our function, we get the type of the function in return:

```
type Fun = GetReturn<typeof createUser>
```

Now we have the ability to infer types that are in this `extends` clause. This happens with the `infer` keyword. We can choose a type variable and return this type variable.

```
type GetReturn<Fun> =
  Fun extends (...args: any[]) => infer R ? R : never
```

With `infer R` we say that no matter the return type of this function, we store it in the type variable `R`. If we have a valid function, we return `R` as type.

```
/*
type User = {
  userId: number,
  name: string,
  role: 'admin' | 'maintenance' | 'shipping',
  isActive: boolean,
  createdAt: Date
}
*/
type User = GetReturn<typeof createUser>
```

Zero maintenance; always correct types. This helper type is available in TypeScript as `ReturnType<Fun>`.

Helper Types

Helper types like `ReturnType` are essential if we construct functions and libraries where we care more about the behavior and interconnection of parts rather than the actual types themselves. Storing and retrieving objects from a database, creating objects based on a schema, that kind of thing. With the `infer` keyword we get powerful flexibility to get types even when we don't know yet what we are dealing with.

For example, a simple type that allows us to retrieve the resolved value of a promise:

```
type Unpack<T> =
  T extends Promise<infer Res> ? Res : never

type A = Unpack<Promise<number>> // A is number
```

Or a type that flattens an array, so we get the type of the array's values.

```
type Flatten<T> =
  T extends Array<infer Vals> ? Vals : never

type A = Flatten<Customer[]> // A is Customer
```

TypeScript has a couple more built-in conditional types that use inference. One is `Parameters`, which collects all arguments from a function in a tuple.

```
type Parameters<T> =
  T extends (...args: infer Param) => any
    ? Param
    : never

/* A is [
    string,
    "admin" | "maintenace" | "shipping",
    boolean
  ]
```

```
*/
type A = Parameters<typeof createUser>
```

Others are:

- `InstanceType`. Gets the type of the created instance of the class's constructor function.
- `ThisParameterType`. If you use callback functions that bind `this`, you can get the bound type in return.
- `OmitThisParameterType`. Uses `infer` to return a function signature without the `this` type. This is handy if your app doesn't care about the bound `this` type and needs to be more flexible in passing functions.

Types with the `infer` keyword have one thing in common: they are low-level utility types that help you glue parts of your code together with ease. This is behavior defined in a type, and it allows for very generic scenarios where your code has to be flexible enough to deal with unknown expectations.

Lesson 42: Working with null

In chapter 4 we learned that `undefined` and `null` are parts of every set in the type space, unless we set the flag

`strictNullChecks`. This removes `undefined` and `null` and treats them as their own entities. This prompts TypeScript to throw red squigglies at us the moment we forget to handle nullish values.

This has a great effect for the code we write on our own. If we use types as contracts to pass data across functions, we can be sure that we have dealt with `null` and `undefined` already. The bitter truth is that nullish values can and will happen, at least at the point where our software has to work with external input:

1. an element from the DOM we want to select
2. user input from an input field
3. data we fetch asynchronously from a back end

Let's look at a very simplified `fetchOrderList` function that does roughly the same thing as the one earlier in this chapter, but which is exclusively asynchronous.

```
declare function fetchOrderList(
  input: Customer | Product
): Promise<Order[]>
```

This function's contract tells us that we get a promise that definitely returns a list of orders. The implicit message is

that fetching orders has also succeeded, and that handling errors or nullish values has already happened.

If we implement this function with `fetch` as we did in lesson 37, we see that we have a problem: `fetch` returns a promise of `any` (the top type that covers everything and takes anything, including `null` and `undefined` – and `never`, if we take the possibility of an error into account).

This means that inside this function, we lose the information if the return value is actually defined. We have to be more specific about the set of possible return values, especially since `strictNullChecks` says we don't take nullish values into our sets.

The *real* function head for `fetchOrderList` is much more like this:

```
declare function fetchOrderList(
  input: Customer | Product
): Promise<Order[] | null>
```

This is good. We add nullish values back to our sets and are explicit about it. This means that we are also forced to check if values can be null. This makes our code much safer than before.

NonNullable

To handle `null`, we have two possibilities, as shown by a function called `listOrders` that prints all orders on the console.

The first option is that we add `null` to the input arguments.

```
declare function listOrders(Order[] | null): void
```

This ensures that the `listOrders` function is compatible with the output from `fetchOrderList`. Null checks have to be done inside `listOrders`. The other option is to make sure that we never pass nullish values to `listOrders`. This means that we have to do null checks before:

```
declare function listOrders(Order[]): void
```

In any case, we will have to do a check for `null`. And most likely not only for lists of orders, but also for lists of products, lists of customers, and so on. This calls for a generic helper function, that checks if an object is actually available.

```
declare function isAvailable<Obj>(
  obj: Obj
): obj is NonNullable<Obj>
```

This generic function binds to the type variable `Obj`. So far, `Obj` can be anything. We don't have any type constraints and don't want any type constraints. `isAvailable` should work with everything.

But the result should ensure we don't have any nullish values. That's why we use the built-in utility type `NonNullable`, which removes `null` and `undefined` from our set of values.

`NonNullable` is defined as follows:

```
type NonNullable<T> =
  T extends null | undefined ? never : T
```

`NonNullable` is a distributive conditional type. If we pass a union where we explicitly set `null` or `undefined`, we can make sure that we remove these value types again and continue with a narrowed down set. This is the implementation:

```
function isAvaialble<Obj>(
  obj: Obj
): obj is NonNullable<Obj> {
  return typeof obj !== 'undefined' &&
    obj !== null
}
```

This is the helper function in action:

```
// orders is Order[] | null
const orders = await fetchOrderList(customer)
if(isAvailable(orders)) {
  //orders is Order[]
  listOrders(orders)
}
```

It's recommended to consider nullish values early on. Keep the core of your application null-free, and try to catch any possible side effects as soon as possible.

Low-Level Utilities

TypeScript's built-in conditional types help a lot if you work on very low-level utility functions that you can reuse in your application.

The same goes for our own utility types we declared in the previous lesson. `fetchOrderList` is very specific; now think of a much more generic function and about some possible processes.

First, fetching from a database.

```
type FetchDBKind =
  'orders' | 'products' | 'customers'

type FetchDBReturn<T> =
  T extends 'orders' ?  Order[] :
  T extends 'products' ? Products[] :
  T extends 'customers' ? Customers[] : never

declare function fetchFromDatabase<
  Kin extends FetchKind
>(
  kind: Kin
): Promise<FetchDbReturn<Kin>| null>
```

Processing anything that we fetched and making sure we get the right process function for this.

```
function process<T extends Promise<any>>(
  promise: T,
  cb: (res: Unpack<NonNullable<T>>) => void
): void {}
  promise.then(res =>
    if(isAvailable(res)) {
      cb(res)
    }
  )
}
```

This allows us to safely `listOrders` to a function where the results can be ambiguous.

```
process(
  fetchFromDatabase('orders'),
  listOrders
)
```

Recap

Conditional types are perhaps the most unique thing about TypeScript's type system. And they come as a direct result of the enormous flexibility JavaScript has to offer to developers. This is what we learned.

1. Conditional types are a great tool to make direct connections between input types and output types, something that gets lost in function overloads very early.
2. Still, we need a good and healthy mix of conditional types and function overloads to make sure that functions with ambiguous results can be understood in a type-safe manner.

3. We learned about the distributive nature of conditional types when used with naked type parameters. A conditional type of union types is like a union type of conditional types.
4. This allows us to filter with the `never` type, reducing unions and being more explicit about what we expect.
5. With helper types like `Pick`, `Extract`, and `Exclude` we can model behavior to a set of data, and make sure that no matter how the data changes, our processes are prepared for change. Less type maintenance, more type safety.
6. We learned about the `infer` keyword and how it can be used to extract types from a much more complex generic type.
7. We learned about working with `null` and the `NonNullable` type, and realized how a good set of low-level primitives can make our own code more generic and flexible without losing any type safety.

Conditional types should become invaluable tools on your TypeScript belt. You'll be able to create strong typings with just a few lines of code, and can focus exclusively on writing JavaScript code. As we'll see in the next chapter.

Interlude: lib.d.ts

Because TypeScript is released multiple times per year, and JavaScript has a fixed update once a year, it is almost impossible to keep track of all the built-in utilities and helpers provided by TypeScript itself.

One good source of type definitions comes through looking at the source itself! TypeScript comes with a lot of library code, defining the JavaScript essentials we can't work without: the `Array` object, the `Object` object. `Number`, `String` – you name it.

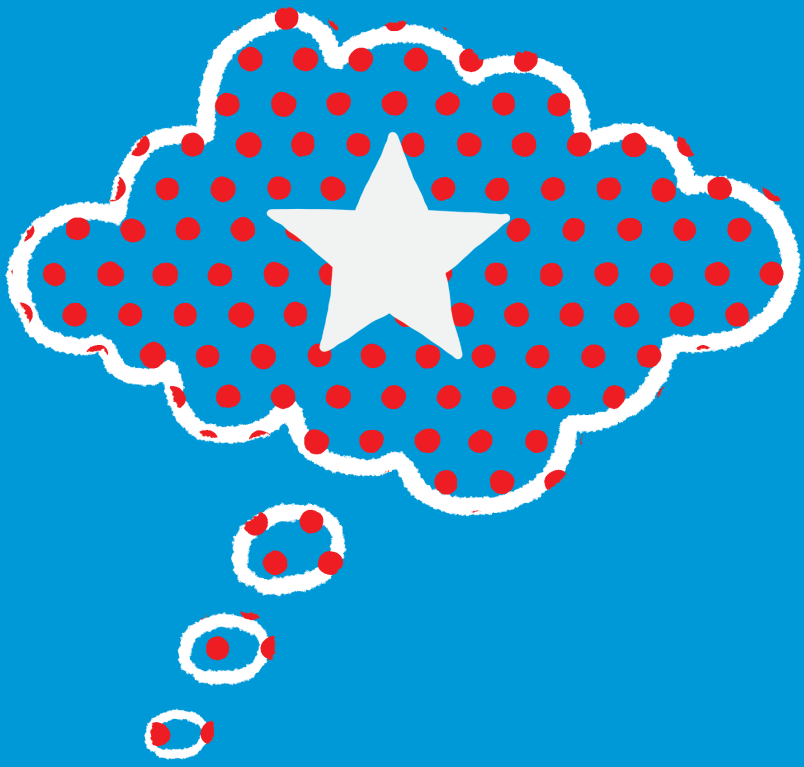
To access the library, open the *lib* folder in your TypeScript installation. Or quicker: select **Go to definition** when you right-click on a built-in type in your editor. This will open the respective file, and also show you an extensive list of files that are split by version and environment.

Those types make interesting reading. You will see what's defined in the DOM; you will see language features by ECMAScript version. This lets you see how the TypeScript team uses TypeScript to define what's happening in JavaScript. It also lets you see the behavior of new ECMAScript features in a very condensed but extremely well-documented manner.

For example, in ES2018 we got a `finally` clause for promises. It allows you to specify a task that is executed no matter if the promise resolves to a value or is rejected. This is what the *lib.es2018.promise.d.ts* file has to say:

```
/**
 * Represents the completion of an asynchronous
operation
 */
interface Promise<T> {
    /**
     * Attaches a callback that is invoked when
     * the Promise is settled (fulfilled or rejected).
     * The resolved value cannot be modified from the
     * callback.
     * @param onfinally The callback to execute when
     *   the Promise is settled (fulfilled or
     *   rejected).
     * @returns A Promise for the completion of the
     *   callback.
     */
    finally(onfinally?: (() => void) |
      undefined | null) : Promise<T>
}
```

Due to declaration merging, this small snippet is added to the complete specification of `Promise`. That's how we don't have to read the full specification but can focus on what's new. Handy!



Chapter Seven

THINKING IN TYPES

Thinking in Types

We are now at the point in our TypeScript journey where we have a good understanding of how the type system works. We started with simple types for our objects and functions. We learned more about widening and narrowing types within the type space. And with generics and conditional types, we prepared ourselves for both unknown and modeled behavior.

These are quite some tools we have to make our JavaScript code more type-safe, more robust, and less error-prone. In this chapter, we'll work to strengthen our knowledge by seeing solutions to problems you might encounter every day in your TypeScript life. Our goal is to write just a couple of types to make our life easier, so we can focus on coding more JavaScript.

And along the way we'll learn some new concepts!

Lesson 43: Promisify

In this lesson we'll use:

- union types for function overloads
- conditional types with `infer`
- variadic tuple types

Once you get used to writing promises, it becomes harder and harder to go back to the old-style callback way of asynchronous programming. If you are anything like me, the first thing you'd do is wrap all callback-style functions in a promise and move along. For me, this has become a pattern I use so often that I wrote my own `promisify` function: a function that takes any other callback-based function and creates a promise-style function out of it.

This is what we expect as behavior:

```
// e.g. a function that loads a file into a string
declare function loadFile(
  fileName: string,
  cb: (result: string) => void
);

// We want to promisify this function
const loadFilePromise = promisify(loadFile)

// The promised function takes all arguments
// except the last one and returns a promise
// with the result
loadFilePromise('./chapter7.md')
  // which is a string according to loadFile
  .then(result => result.toUpperCase())
```

When thinking in types, we always start by declaring a function and making sure we know what to expect as input and output. The input is a function that can be promisified,

which means it has a callback at the end of the function. The output is a promisified function, which means a function that takes a number of arguments and returns a promise.

A function with a callback as input, and a promisified function as output. There are our input and output types:

```
declare function promisify<
  Fun extends FunctionWithCallback
>(fun: Fun): PromisifiedFunction<Fun>;
```

Now, let's model our newly created types. `FunctionWithCallback` is somehow peculiar, as it needs to work for a potentially endless list of arguments before we reach the last one, the callback.

One way to achieve that would be a list of function overloads, where we make sure the last argument is a callback. But this method has to end somewhere. The following example stops at three overloads:

```
type FunctionWithCallback =
  (
    (
      arg1: any,
      cb: (result: any) => any
    ) =>  any
  ) |
```

```
(
  (
    arg1: any,
    arg2: any, cb: (result: any) => any
  ) : any
) |
(
  (
    arg1: any,
    arg2: any,
    arg3: any,
    cb: (result: any) => any
  ) : any
)
```

A much more flexible solution is a *variadic tuple type*. A tuple type in TypeScript is an array with the following features.

1. The length of the array is defined.
2. The type of each element is known (and does not have to be the same).

For example, this is a tuple type:

```
type PersonProps = [string, number]

const [name, age]: PersonProps = ['Stefan', 37]
```

Function arguments can also be described as tuple types. For example:

```
declare function hello(name: string, msg: string):
void;
```

is the same as:

```
declare function hello(...args: [string, string]):
void;
```

A variadic tuple type is a tuple type that has the same properties – defined length and the type of each element is known – but where the *exact shape is yet to be defined*.

This is a perfect use case for our callback-style function. The last argument has to be a callback function; everything before is yet to be defined. So `FunctionWithCallback` can look like this:

```
type FunctionWithCallback<T extends any[] = any[]> =
  (...t: [...T, (...args: any) => any]) => any;
```

First, we define a generic. We need generics to define variadic tuple types as we have to define the shape later on. This generic type parameter extends the `any[]` array to catch all tuples. We also default to `any[]` to make it easier to use. Then comes a function definition.

The argument list is of type `[...T, (...args: any) => any])`. First the *variadic* part that we define through usage, then a wildcard function. This ensures we only pass functions that have a callback as the last argument.

Note that we explicitly use `any` here. This is one of the rare use cases where `any` makes a lot of sense. We're not concerned yet about what we're passing as a function as long as the shape is intact. We also don't want to be bothered by passing arguments around. This is a helper function and `any` will do just fine.

Next, let's work on the return type, a promisified function. The promisified function is a conditional type that checks the shape we defined in `FunctionWithCallback`. This is where the actual type check happens, and where we bind types to generics.

```
type PromisifiedFunction<T> =
  (...args: InferArguments<T>)
    => Promise<InferResults<T>>
```

A `PromisifiedFunction<T>` takes all the arguments of the original callback-style function minus the callback. We get this argument list through `InferArguments<T>`.

```
type InferArguments<T> =
  T extends (
    ... t: [...infer R, (...args: any) => any]
  ) => any ? R : never
```

The conditional checks against the same type declaration we defined in `FunctionWithCallback`, but instead of letting the arguments before the callback just pass, we infer the whole list of arguments and return them.

The `PromisifiedFunction<T>` returns a promise with the results defined in the callback function. That's why we have to *infer* the results from the original function in the same manner:

```
type InferResults<T> =
  T extends (
    ...t: [...infer T, (res: infer R) => any]
  ) => any ? R : never
```

The function type we check for in our conditional is again of a very similar shape to `FunctionWithCallback`. This

time, however, we want to know the argument list of the callback and infer those.

This already works like a charm. `loadFile` gets the correct types and also functions with other types, and other argument lists do the same.

```
declare function addAsync(
  x: number, y: number,
  cb: (result: number) => void
);

const addProm = promisify(addAsync);
// x is number!
addProm(1, 2).then(x => x)
```

The types are done! We can test how this function will behave once it's finished. And this is all we need to do on the TypeScript side. A couple of lines of code and we know the input and the output behavior of our function.

As this is a utility function, we will most likely use `promisify` more than once. Time on typings well spent.

The implementation also takes a couple of lines of code:

```
function promisify<
  Fun extends FunctionWithCallback
>(f: Fun): PromisifiedFunction<Fun> {
  return function(...args: InferArguments<Fun>) {
    return new Promise((resolve) => {
      function callback(result: InferResults<Fun>) {
        resolve(result)
      }
      args.push(callback);
      f.call(null, ...args)
    })
  }
}
```

Here we can see some behavior mirrored in JavaScript that we already saw in TypeScript. We use rest parameters – of type `InferArguments` – in the newly created function. This returns a promise with a result of type `InferResults`. To get this, we need to create the callback, let it *resolve* the promise with the results and add it back to the argument lists. Now we can call the original function with a complete set of arguments.

This is the inverse operation of what we did with our types, where we always tried to shave off the callback function. Here we need to add it again to make it complete. The types are complex, but so sound we can implement the whole `promisify` function without any type cast. This is a good sign!

Lesson 44: JSONify

In this lesson we'll see:

- mapped types
- conditional types
- the `infer` keyword
- recursive types

Hat tip to Anders Hejlsberg. This is one of the examples he showcased in one of his talks. I spent quite some time scribbling it down from pixelated screengrabs, and even more time figuring out what was happening in just a couple lines of code. After seeing the power and potential of the type system, this was the moment where I became a TypeScript fan.

`JSON.parse` and `JSON.stringify` are nice functions to serialize and deserialize JavaScript objects. JSON is a subset of JavaScript objects, missing only functions and `undefined`. This subset makes parsing JSON strings even faster than parsing regular objects.[29]

A class that serializes and deserializes JavaScript objects, which can be bound to different types, could look like this:

```
class Serializer<T> {
```

29 https://smashed.by/jsoncost

```
  serialize(inp: T): string {
    return JSON.stringify(inp)
  }

  deserialize(inp: string): JSONified<T> {
    return JSON.parse(inp)
  }
}
```

With `JSONified` to be defined. Let's declare a type that includes all possible ways of writing values in JavaScript: numbers, strings, Booleans, functions. Nested and in arrays. And a type that has a `toJSON` function. If an object has a `toJSON` function, `JSON.stringify` will use the object returned from `toJSON` for serialization and not the actual properties.

```
// toJSON returns this object for
// serialization, no matter which other
// properties this type has.
type Widget = {
  toJSON(): {
    kind: "Widget", date: Date
  }
}

type Item = {
  // Regular primitive types
  text: string;
  count: number;
  // Options get preserved
```

```
  choice: "yes" | "no" | null;
  // Functions get dropped.
  func: () => void;
  // Nested elements need to be parsed
  // as well
  nested: {
    isSaved: boolean;
    data: [1, undefined, 2];
  }
  // A pointer to another type
  widget: Widget;
  // The same object referenced again
  children?: Item[];
}
```

There is a difference between JSON and JavaScript objects, and we can model this difference with just a few lines of conditional types. Let's implement `JSONified<T>`.

JSONified Values

First, we create the `JSONified` type and do one particular check: is this an object with a `toJSON` function? If so, we infer the return type and use it. Otherwise we JSONify the object itself. `toJSON` also returns an object, so we pass it to our next step as well.

```
type JSONified<T> =
  JSONifiedValue<
    T extends { toJSON(): infer U } ? U : T
  >;
```

Next, let's look at the actual values, and what happens once we serialize them. Primitive types can be transferred easily. Functions should be dropped. Arrays and nested objects should be handled separately.

Let's put this into a type:

```
type JSONifiedValue<T> =
  T extends string | number | boolean | null ? T :
  T extends Function ? never :
  T extends object ? JSONifiedObject<T> :
  T extends Array<infer U> ? JSONifiedArray<U> :
  never;
```

`JSONifiedObject` is a mapped type where we run through all properties and apply `JSONified` again.

```
type JSONifiedObject<T> = {
  [P in keyof T]: JSONified<T[P]>
}
```

This is the first occurence in this book of a *recursive* type. TypeScript allows for a certain level of recursion as long as there's no circular referencing involved. Calling `JSONified` again further down a tree works.

It's similar with the `JSONifiedArray`, which becomes an array of JSONified values. If there's an `undefined` element, `JSON.stringify` will map this to `null`. That's why we need another helper type.

```
type UndefinedAsNull<T> =
  T extends undefined ? null : T;

type JSONifiedArray<T> =
  Array<UndefinedAsNull<JSONified<T>>>
```

And this is all we need. With a couple lines of code, we described the entire behavior of a `JSON.parse` after a `JSON.stringify`. Not only on a type level, but also wrapped nicely in a class:

```
const itemSerializer = new Serializer<Item>()

const serialization =
  itemSerializer.serialize(anItem)

const obj =
  itemSerializer.deserializer(serialization)
```

I urge you to try it out in a playground or in your own application. It's remarkable to see how recursive types work and how deep they can go in a nested object structure.

Lesson 45: Service Definitions

In this lesson we'll see:

- mapped types
- conditional types
- `String` and `Number` constructors
- control flow analysis

Another hat tip to Anders Hejlbserg and the TypeScript team. This is the second example that made me love TypeScript and acknowledge the immense power that lies in the type system.

Dynamic Definitions

We want to provide a helper function for our colleagues so they can define a service and its service methods through a JavaScript object that looks a little like a type; for example, a service definition for opening, closing, and writing to files.

```
const serviceDefinition = {
  open: { filename: String },
  insert: { pos: Number, text: String },
  delete: { pos: Number, len: Number },
  close: {},
};
```

The service definition object describes a list of method definitions. Each property of the service definition is a method. Each method definition defines a payload and the accompanying type.

We use capital `Number` and capital `String` here, constructor functions in JavaScript that create values of type `number` or `string` respectively. These are not TypeScript types! But they look awfully similar.

The goal is to have a function, `createService`, where we pass two things:

3. The **service definition**, in the format we described above.
4. A **request handler**. This is a function that receives messages and payloads, and is implemented by the users of this function; for example, for the service definition above we expect to get a message `open` with a payload `filename`, where the file name is a string

In return, we get a service object. This service object exposes methods (open, insert, delete, and close, according to the service definition) that allow for a certain payload (defined in the service definition). Once we call a method, we set up a request object that is handled by the request handler.

Typing Service Definitions

As always, before we implement something, let's work on the function head of `createService`. According to the specification above, the plain function definition looks like this:

```
declare function createService<
  S extends ServiceDefinition
>(
  serviceDef: S,
  handler: RequestHandler<S>,
): ServiceObject<S>
```

Note that we only need one bound generic type variable: the service definition. Let's define the service definition types.

```
// A service definition has keys we don't know
// yet and lots of method definitions
type ServiceDefinition = {
  [x: string]: MethodDefinition;
};
```

```
// This is the payload of each method:
// a key we don't know, and either a string or
// a number constructor (the capital String, Number)
type MethodDefinition = {
  [x: string]:
    StringConstructor | NumberConstructor;
};
```

This allows for objects with every possible string key. The moment we bind a concrete service definition to the generic variable, the keys become defined, and we work with the narrowed-down type. Next, we work on the second argument, the request handler. The request handler is a function with one argument, the request object. It returns a Boolean if the execution was successful.

```
type RequestHandler<
  T extends ServiceDefinition
> = (req: RequestObject<T>) => boolean;
```

The Request Object

The request object is defined by the service definition we pass. It's an object where each key of the service definition

becomes a message. The object after the key of the service definition becomes the payload.

```
type RequestObject<T extends ServiceDefinition> = {
  [P in keyof T]: {
    message: P;
    payload: RequestPayload<T[P]>;
  }
}[keyof T];
```

With the index access type to `keyof T`, we create a union out of an object that would contain every key.

The request payload is defined by the object of each key in the service definition:

```
type RequestPayload<T extends MethodDefinition> =
  // Is it an empty object?
  {} extends T ?
    // Then we don't have a payload
    undefined :
    // Otherwise we have the same keys, and get the
    // type from the constructor function
    { [P in keyof T]: TypeFromConstructor<T[P]> };

type TypeFromConstructor<T> =
  T extends StringConstructor ?
    string :
    T extends NumberConstructor ? number : any;
```

For the service definition we described earlier, the generated type looks like this:

```
{
  req: {
    message: “open”;
    payload: {
        filename: string;
    };
  } | {
    message: “insert”;
    payload: {
        pos: number;
        text: string;
    };
  } | {
    message: “delete”;
    payload: {
        pos: number;
        len: number;
    };
  } | {
    message: “close”;
    payload: undefined;
  }
}
```

The Service Object

Last, but not least, we type the service object, the return value. For each entry in the service definition, it creates some service methods.

```
type ServiceObject<T extends ServiceDefinition> = {
  [P in keyof T]: ServiceMethod<T[P]>
};
```

Each service method takes a payload defined in the object after each key in the service definition.

```
type ServiceMethod<T extends MethodDefinition> =
  // The empty object?
  {} extends T ?
  // No arguments!
  () => boolean :
  // Otherwise, it's the payload we already
  // defined
  (payload: RequestPayload<T>) => boolean;
```

That's all the type we need!

Implementing createService

Now, let's implement the function itself. We create an empty object and add new keys to it. Each key is a method that takes a payload and calls the handler.

```
function createService<S extends ServiceDefinition>(
  serviceDef: S,
  handler: RequestHandler<S>,
): ServiceObject<S> {
  const service: Record<string, Function> = {};

  for (const name in serviceDef) {
    service[name] =
      (payload: any) => handler({
        message: name,
        payload
      });
  }

  return service as ServiceObject<S>;
}
```

Note that we allow ourselves a little type cast at the end to make sure the very generic creation of a new object is type-safe. This is how our service definition looks in action:

```
const service = createService(
  serviceDefinition,
  req => {
```

```
    // req is now perfectly typed and we know
    // which messages we are able to get
    switch (req.message) {
      case 'open':
        // Do something
        break;
      case 'insert':
        // Do something
        break;
      default:
        // Due to control flow anaysis, this
        // message now can only be close or
        // delete.
        // We can narrow this down until we
        // reach never
          break;
    }
    return true;
  }
);
// We get full autocomplete for all available
// methods, and know which payload to
// expect
service.open({ filename: 'text.txt' });

// Even if we don't have a payload
service.close();
```

We wrote around 25 lines of type definitions and a few implementation details, and our colleagues will be able to

write custom services that are completely type-safe. Take any other service definition and play around yourself!

Lesson 46: DOM JSX Engine, Part 1

In this lesson we'll learn about JSX and JSX factories.

Over the last couple of years, there has hardly been a technology more discussed than React. Well, maybe Typescript! React is a library to compose UI and handle state directly in JavaScript – no other technology needed. Except one: JSX. JSX is a syntax feature that allows you to write HTML or XML-like commands directly in JavaScript. Your components look something like this:

```
export function Button() {
  return <button>Click me</button>
}
```

If you'd never seen something like this, and even if you had, your first reaction might be: "JSX mixes HTML with my JavaScript – that's ugly!" Believe me, that was *my* first reaction. Rest assured, JSX doesn't mix HTML and JavaScript. Here's what JSX is not:

- JSX is not a templating language
- JSX is not HTML
- JSX is not XML

JSX *looks* like all that, but it's nothing but syntactic sugar.

JSX Is Function Calls

JSX translates into pure, nested function calls. The React method signature of JSX is `(element, properties, ...children)`, with `element` being either a React component or a string, `properties` being a JS object with keys and values, and `children` being empty, or an array with more function calls.

So:

```
<Button onClick={() => alert('YES')}>Click me</Button>
```

translates to:

```
React.createElement(Button, { onClick: () =>
alert('YES') }, 'Click me');
```

With nested elements, it looks something like this:

```
<Button onClick={() => alert('YES')}><span>Click me</
span></Button>
```

which translates to:

```
React.createElement(Button, { onClick: () =>
alert('YES') },
  React.createElement('span', {}, 'Click me'));
```

There is one convention: uppercase elements translate to components, lowercase elements to strings. The latter is used for standard HTML elements.

What are the implications, especially compared with templates?

- There's no runtime compilation and parsing of templates. Everything goes directly to the virtual DOM or layout engine underneath.
- There are no expressions to evaluate. Everything around is JavaScript.
- Every component property is translatable to a JSX object key. This allows us to type check them.

> TypeScript works so well with JSX because there's JavaScript underneath.

So everything looks like XML, except that it's JavaScript functions.

One question to we seasoned web developers: have you ever wanted to write to the DOM directly, but you gave up because it's so unwieldy? `document.createElement` has an easy enough API, but we have to do a ton of calls to the DOM API to get what we can achieve so easily by writing HTML. JSX solves that. With JSX you have a nice and familiar syntax of writing elements without HTML.

Writing the DOM with JSX

Enter TypeScript! TypeScript is a full-blown JSX compiler. With TypeScript we have the possibility to change the JSX factory. That's how TypeScript is able to compile JSX for React, Vue.js, Dojo... any framework using JSX in one way or another. The virtual DOM implementations underneath might differ, but the interface is the same:

```
/**
 * element: string or component
 * properties: object or null
 * ...children: null or calls to the factory
 */
```

```
function factory(element, properties, ...children) {
  //...
}
```

We can use the same factory method signature not only to work with the virtual DOM, but also to work with the real DOM, only to have a nice API on top of `document.createElement`.

Let's try! These are the features we want to implement:

1. Parse JSX to DOM nodes, including attributes.
2. Have simple functional components for more composability and flexibility.

TypeScript needs to know how to compile JSX for us. Setting two properties in *tsconfig.json* is all we need.

```
{
  "compilerOptions": {
    ...
    "jsx": "react",
    "jsxFactory": "DOMcreateElement",
    "noImplicitAny": false
  }
}
```

We leave it to the React JSX pattern (the method signature we were talking about earlier), but tell TypeScript to use our soon-to-be-created function `DOMcreateElement` for that.

Also, we set `noImplicitAny` to `false` for now. This is so we can focus on the implementation and do proper typings at a later stage. If we want to use JSX, we have to rename our *.ts* files to *.tsx*.

First, we implement our factory function. Its specification:

1. If the element is a function, then it's a functional component. We call this function (passing `properties` and `children`, of course) and return the result. We expect a return value of type `Node`.
2. If the element is a string, we parse a regular node.

```
function DOMcreateElement(
  element, properties, ...children
) {
  if(typeof element === 'function') {
    return element({
      ...nonNull(properties, {}),
      children
    });
  }
```

```
  return DOMparseNode(
    element,
    properties,
    children
  );
}

/**
 * A helper function that ensures we won't work with
null values
 */
function nonNull(val, fallback) { #
  return Boolean(val) ? val : fallback
};
```

Next, we parse regular nodes.

1. We create an element and apply all properties from JSX to this DOM node. This means that all properties we can pass are part of `HTMLElement`.
2. If available, we append all children.

```
function DOMparseNode(element, properties, children) {
  const el = Object.assign(
    document.createElement(element),
    properties
  );
  DOMparseChildren(children).forEach(child => {
```

```
    el.appendChild(child);
  });
  return el;
}
```

Last, we create the function that handles `children`. Children can either be:

1. Calls to the factory function `DOMcreateElement`, returning an `HTMLElement`.
2. Text content, returning a `Text`.

```
function DOMparseChildren(children) {
  return children.map(child => {
    if(typeof child === 'string') {
      return document.createTextNode(child);
    }
    return child;
  })
}
```

To sum it up:

1. The factory function takes elements. Elements can be of type `string` or a function.

2. A function element is a component. We call the function, because we expect to get a DOM `Node` out of it. If the function component has also more function components inside, they will eventually resolve to a DOM `Node` at some point.
3. If the element is a string, we create a regular DOM `Node`. For that we call `document.createElement`.
4. All properties are passed to the newly created `Node`. By now you might understand why React has something like `className` instead of `class`. This is because the DOM API underneath is also `className`. `onClick` is camelCase, though, which I find a little odd.
5. Our implementation only allows DOM `Node` properties in our JSX, because of that simple property passing.
6. If our component has children (pushed together in an array), we parse children as well and append them.
7. Children can be either a call to `DOMcreateElement`, resolving in a DOM `Node` eventually, or a simple string.
8. If it's a string, we create a `Text`. `Text`s can also be appended to a DOM `Node`.

That's all there is! Look at the following code example:

```
const Button = ({ msg }) => {
  return <button onclick={() => alert(msg)}>
```

```
    <strong>Click me</strong>
  </button>
}

const el = (
  <div>
    <h1 className="what">Hello world</h1>
    <p>
      Lorem ipsum dolor sit, amet consectetur
      adipisicing elit. Quae sed consectetur
      placeat veritatis
      illo vitae quos aut unde doloribus, minima
eveniet et
      eius voluptatibus minus aperiam
      sequi asperiores, odio ad?
    </p>
    <Button msg='Yay' />
    <Button msg='Nay' />
  </div>
)

document.body.appendChild(el);
```

Our JSX implementation returns a DOM `Node` with all its children. We can even use function components for it. Instead of templates, we work with the DOM directly, but the API is a lot nicer!

So what's missing? Property typings!

Lesson 47: DOM JSX Engine, Part 2

In this lesson we'll see:

- type maps
- mapped types
- conditional types
- declaration merging
- the JSX namespace

What we created in the previous lesson is already *very* powerful. We can go really far and have a nice API to write to the DOM without any library or framework!

But we TypeScript folks miss one important thing: proper typings. Type inference does a lot for us, but with `noImplicitAny` turned off, we miss a lot of important information.

Turning it back on, we see red squigglies everywhere. Let's be true to ourselves and create proper typings for our functions. As an added benefit, we also get a really good type inference!

Typing the Factory

Let's start with the three functions that we have. TypeScript at this point really complains mostly about implicit `any`s. So we'd better throw in some concrete types. For brevity, we'll just focus on the function heads.

The `nonNull` helper function is easy to type. We take two generics we can bind as we use the function.

```
function nonNull<T, U>(val: T, fallback: U) {
  return Boolean(val) ? val : fallback;
}
```

The return type is inferred as `T | U`.

Next, we work on `DOMparseChildren` as it has the simplest set of arguments. There are just a few types that can actually be children of our DOM tree:

1. `HTMLElement`, the base class of all HTML elements.
2. `string`, if we pass a regular string that should be converted into a text node.
3. `Text`, if we created a text node outside that we want to append.

We create a helper union, `PossibleChildren`, and use this for the argument of `DOMparseChildren`.

```
type PossibleChildren =
  HTMLElement | Text | string

function DOMparseChildren(
  children: PossibleChildren[]
) {
  // ...
}
```

The return type is correctly inferred as `HTMLElement | Text`, as we get rid of `string` and convert them to `Text`. The next function is `DOMparseNode`, as it has the same signature as `DOMcreateElement`. Let's look at the possible input arguments.

1. `element` can be a string or a function. We want to use a generic to bind the concrete value of an element to a type.
2. `properties` can be either the function arguments of the component function or the set of properties of the respective HTML element.
3. `children` is a set of possible children. We have the type for this already.

To make this work properly, we need a couple of helper types. `Fun` is a much looser interpretation of `Function`. We need this to infer parameters.

```
type Fun =  (...args: any[]) => any;
```

We need to know which HTML element is created when we pass a certain string. TypeScript provides an interface called `HTMLElementTagNameMap`. It's a so-called type map, which means that it is a key–value list of identifiers (tags) and their respective types (subtypes of `HTMLElement`). You can find this list in *lib.dom.ts*.

```
interface HTMLElementTagNameMap {
  "a": HTMLAnchorElement;
  "abbr": HTMLElement;
  "address": HTMLElement;
  "applet": HTMLAppletElement;
  // and so on ...
}
```

We want to create a type, `CreatedElement`, that returns the element according to the string we pass. If this element doesn't exist, we return `HTMLElement`, the base type.

```
type AllElementsKeys = keyof HTMLElementTagNameMap
```

```
type CreatedElement<T> =
  T extends AllElementsKeys ? HTMLElementTagNameMap[T]
:
  HTMLElement
```

We use this helper type to properly define `Props`. If we pass a function, we want to get parameters of this function. If we pass a string, possible properties are a partial of the corresponding HTML element. Without the partial, we would have to define *all* properties. And there are a lot!

```
type Props<T> =
  T extends Fun ? Parameters<T>[0] :
  T extends string ? Partial<CreatedElement<T>> :
  never;
```

Note that we index the first parameter of `Parameters`. This is because the JSX function only has one argument, the props. And we need to destructure from a tuple to an actual type.

This is all we need for now. Let's go for `DOMparseNode`. This function only works if we pass strings.

```
function DOMparseNode<
```

```
  T extends string
>(
  element: T,
  properties: Props<T>,
  children: PossibleElements[]
)
```

The function correctly infers properties of `HTMLElement` as return type. Last, but not least, the `DOMcreateElement` function. This one can be a bit tricky as separating between function properties and HTML element properties is not as easy as it looks at first. Our best option is function overloads as we only have two variants. Also, the `Props` type helps us make the correct connection between the type of `element` and the respective properties.

```
function DOMcreateElement<
  T extends string
>(
  element: T, properties: Props<T>,
  ...children: PossibleElements[]
): HTMLElement
function DOMcreateElement<
  F extends Fun
>(
  element: F, properties: Props<F>,
  ...children: PossibleElements[]
): HTMLElement
```

```
function DOMcreateElement(
  element: any, properties: any,
  ...children: PossibleElements[]
): HTMLElement {
  // ...
}
```

We can now use the factory function directly!

Typing JSX

We still get some type errors when using JSX instead of factory functions. This is because TypeScript has its own JSX parser and wants to catch JSX problems early on. Right now, TypeScript doesn't know which elements to expect. Therefore, it defaults to `any` for every element.

To change this, we have to extend TypeScript's own JSX namespace and define the return type of created elements. Namespaces are a way in TypeScript to organize code. They were created in a time before ECMAScript modules and are therefore not used as much anymore.

Still, when defining internal interfaces that should be grouped, namespaces are the way to go.

As with interfaces, namespaces allow for declaration merging. We open the namespaces JSX and define two things:

1. The return element, which is extending the interface `Element`.
2. All available HTML elements, so TypeScript can give us autocompletion in JSX. They are defined in `IntrinsicElements`. We use a mapped type where we copy `HTMLElementTagNameMap` to be a map of partials. Then we extend `IntrinsicElements` from it. We want to use an interface instead of a type as we want to keep declaration merging open.

```
// Open the namespace
declare namespace JSX {
  // Our helper type, a mapped type
  type OurIntrinsicElements = {
    [P in keyof HTMLElementTagNameMap]:
      Partial<HTMLElementTagNameMap[P]>
  }

  // Keep it open for declaration merging
  interface IntrinsicElements
    extends OurIntrinsicElements {}

  // JSX returns HTML elements. Keep this also
  // open for declaration merging
  interface Element extends HTMLElement {}
}
```

With that, we get autocompletion for HTML elements and function components. And our little TypeScript-based DOM JSX engine is ready for prime time!

Lesson 48: Extending Object, Part 1

This lesson includes:

- conditional types
- mapped types
- ambient declaration files
- declaration merging
- custom type predicates
- constructor interfaces

Hat tip to Mirjam Bäuerlein, who spent an enormous amount of time preparing and discussing this example with me.

TypeScript's control flow analysis lets you narrow down from a broader type to a narrower type:

```
function print(msg: any) {
  if(typeof msg === 'string') {
```

```
    // We know msg is a string
    console.log(msg.toUpperCase()) // thumbs up!
} else if (typeof msg === 'number') {
    // I know msg is a number
    console.log(msg.toFixed(2)) // thumbs up!
  }
}
```

This is a type-safety check in JavaScript, and TypeScript benefits from that. However, there are some cases where TypeScript needs a little bit more assistance from us.

Checking Object Properties

Let's assume you have a JavaScript object and you don't know if a certain property exists. The object might be `any` or `unknown`. In JavaScript, you would check for properties like this:

```
if(typeof obj === 'object'
    && 'prop' in obj) {
  // It's safe to access obj.prop
  console.assert(typeof obj.prop !== 'undefined')
  // But TS doesn't know :-(
}

if(typeof obj === 'object'
    && obj.hasOwnProperty('prop')) {
  // It's safe to access obj.prop
  console.assert(typeof obj.prop !== 'undefined')
```

```
  // But TS doesn't know :-(
}
```

At the moment, TypeScript isn't able to extend the type of `obj` with a `prop`, even though this works with JavaScript. We can, however, write a little helper function to get correct typings:

```
function hasOwnProperty<
  X extends {}, Y extends PropertyKey
>(
  obj: X, prop: Y
): obj is X & Record<Y, unknown> {
  return obj.hasOwnProperty(prop)
}
```

Let's check out what's happening:

1. Our `hasOwnProperty` function has two generics:
    a. `X extends {}` makes sure we use this method only on objects.
    b. `Y extends PropertyKey` makes sure that the key is either `string | number | symbol`. `PropertyKey` is a built-in type.
2. There's no need to explicitly define the generics as they're inferred by usage.

3. `(obj: X, prop: Y)`: we want to check if `prop` is a property key of `obj`.
4. The return type is a type predicate. If the method returns `true`, we can retype any of our parameters. In this case, we say our `obj` is the original object, with an intersection type of `Record<Y, unknown>`. The last piece adds the newly found property to `obj` and sets it to `unknown`.

In use, `hasOwnProperty` works like this:

```
// person is an object
if(typeof person === 'object'
  // person = { } & Record<'name', unknown>
  // = { } & { name: 'unknown'}
  && hasOwnProperty(person, 'name')
  // Yes! name now exists in person
  && typeof person.name === 'string'
  ) {
    // Do something with person.name,
    // which is a string
  }
```

Extending lib.d.ts

Writing a helper function is a bit on the nose. Why write a helper function that wraps some baked-in functionality only to get better types? We should be able to create those typings directly where they occur.

Thankfully, with declaration merging of interfaces, we are able to do that. Create your own ambient type declaration file and make sure that the TypeScript compiler knows where to find them (`typeRoots` and `types` in *tsconfig.json* are a good start).

In this file, which we can call *mylib.d.ts*, we can add our own ambient declarations, and can extend existing declarations.

We can do so with the `Object` interface. This is a built-in interface for all `Object`s.

```
interface Object {
  hasOwnProperty<
    X extends {},
    Y extends PropertyKey
  >(this: X, prop: Y): this is X & Record<Y, unknown>
}
```

If you think TypeScript ought to have something like that out of the box, then you are right. There might be a good reason type definitions aren't yet shipped like that, but it's good that we are able to extend it to meet our own needs.

Extending the Object Constructor

We get into a similar scenario when working with other parts of `Object`. One pattern that you might encounter a lot is to iterate over an array of object keys, then access these properties to do something with the values.

```
const obj = {
  name: 'Stefan',
  age: 38
}

Object.keys(obj).map(key => {
  console.log(obj[key])
})
```

In strict mode, TypeScript wants to know explicitly what type `key` has, to be sure that this index access works. So we get some red squigglies thrown at us. Well, we should know the type of `key` it is! It's `keyof obj`! This is a good chance to extend the typings for `Object`.

This is how `Object.keys` should behave:

1. If we pass a number, we return an empty array.

2. If we pass an array or a string, we get a string array in return. This string array contains the stringified indices of the input value.
3. If we pass an object, we get the actual keys of this object in return.

The interface to extend is called `ObjectConstructor`.
For classes or class-like structures, TypeScript needs two different interfaces. One is the *constructor interface*, which includes the constructor function and all the static information. The other is the *instance interface*, which includes all the dynamic information per instance.

This divide comes from old JavaScript, when classes were defined as constructor function and prototype, for example:

```
// Static parts --> constructor interface
function Person(name, age) {
  this.name = name
  this.age = age;
}

Person.create(name, age) {
  return new Person(name, age)
}
```

```
// Dynamic parts --> instance interface
Person.prototype.toString() {
  return `My name is ${this.name} and I'm
    ${age} years old`
}
```

In our case, `Object` is the instance interface and `Object Constructor` is the constructor interface. Let's make `Object.keys` stronger:

```
// A utility type
type ReturnKeys<O> =
  O extends number ? [] :
  O extends Array<any> | string ? string[] :
  O extends object ? Array<keyof O> : never

// Extending the interface
interface ObjectConstructor {
  keys<O>(obj: O) : ReturnKeys<O>
}
```

Let's put this into our ambient type declaration file and `Object.keys` will get better type inference immediately.

Lesson 49: Extending Object, Part 2

In this lesson, we'll learn about:

- property descriptors
- conditional types
- assertion signatures

In JavaScript we can define object properties on the fly with `Object.defineProperty`. This is useful if we want your properties to be read-only or similar. Think back to the very first example in this book: a storage object that has a maximum value that shouldn't be overwritten:

```
const storage = {
  currentValue: 0
}

Object.defineProperty(storage, 'maxValue', {
  value: 9001,
  writable: false
})

console.log(storage.maxValue) // 9001

storage.maxValue = 2

console.log(storage.maxValue) // Still 9001
```

`defineProperty` and property descriptors are very complex. They allow us to do everything with properties that is usually reserved for built-in objects. So they're common in larger codebases. TypeScript has a little problem with `defineProperty`:

```
const storage = {
  currentValue: 0
}

Object.defineProperty(storage, 'maxValue', {
  value: 9001,
  writable: false
})

// Property 'maxValue' does not exist on type...
console.log(storage.maxValue)
```

If we don't explicitly type cast, we don't get `maxValue` attached to the type of `storage`. However, for simple use cases, we can help!

Assertion Signatures

In TypeScript 3.7 the team introduced *assertion signatures*. Think of an `assertIsNum` function where we can make sure some value is of type `number`, and otherwise it throws an error. This is similar to the `assert` function in Node.js:

```
function assertIsNum(val: any) {
  if (typeof val !== “number”) {
    throw new AssertionError(“Not a number!”);
  }
}

function multiply(x, y) {
  assertIsNum(x);
  assertIsNum(y);
  // At this point I'm sure x and y are numbers.
  // If one assert condition is not true, this
  // position is never reached.
  return x * y;
}
```

To comply with behavior like this, we can add an assertion signature that tells TypeScript that we know more about the type after this function:

```
function assertIsNum(val: any): asserts val is number
{
  if (typeof val !== “number”) {
    throw new AssertionError(“Not a number!”);
  }
}
```

This works a lot like type predicates, but without the control flow of a condition-based structure like `if` or `switch`.

```
function multiply(x, y) {
  assertIsNum(x);
  assertIsNum(y);
  // Now also TypeScript knows that both x and y are
numbers
  return x * y;
}
```

If we look at it closely, we can see those assertion signatures can change the type of a parameter or variable on the fly. This is just what `Object.defineProperty` does as well.

Custom defineProperty

Disclaimer: The following helper does not aim to be 100% accurate or complete. It might have errors, and it might not tackle every edge case of the `defineProperty` specification. It might, however, handle a lot of use cases well enough. So use it at your own risk!

Just as with `hasOwnProperty` in the last lesson, we create a helper function that mimics the original function signature:

```
function defineProperty<
  Obj extends object,
```

```
  Key extends PropertyKey,
  PDesc extends PropertyDescriptor>
  (obj: Obj, prop: Key, val: PDesc) {
  Object.defineProperty(obj, prop, val);
}
```

We work with three generics:

1. The object we want to modify, of type `Obj`, which is a subtype of `object`.
2. Type `Key`, which is a subtype of `PropertyKey` (built-in), so `string | number | symbol`.
3. `PDesc`, a subtype of `PropertyDescriptor` (built-in). This allows us to define the property with all its features (writability, enumerability, reconfigurability).

We use generics because TypeScript can narrow them down to a very specific unit type. `PropertyKey`, for example, is all numbers, strings, and symbols. But if we use `Key extends PropertyKey`, we can pinpoint `prop` to be of (for instance) type `"maxValue"`. This is helpful if we want to change the original type by adding more properties.

The `Object.defineProperty` function either changes the object or throws an error should something go wrong.

Exactly what an assertion function does. Our custom helper `defineProperty` thus does the same.

Let's add an assertion signature. Once `defineProperty` successfully executes, our object has another property. We'll create some helper types for that. The signature first:

```
function defineProperty<
  Obj extends object,
  Key extends PropertyKey,
  PDesc extends PropertyDescriptor>
-  (obj: Obj, prop: Key, val: PDesc) {
+  (obj: Obj, prop: Key, val: PDesc):
+    asserts obj is Obj & DefineProperty<Key, PDesc> {
  Object.defineProperty(obj, prop, val);
}
```

`obj` then is of type `Obj` (narrowed down through a generic) and our newly defined property.

This is the `DefineProperty` helper type:

```
type DefineProperty<
  Prop extends PropertyKey,
  Desc extends PropertyDescriptor> =
    Desc extends {
      writable: any, set(val: any): any
    } ? never :
    Desc extends {
      writable: any, get(): any
```

```
  } ? never :
  Desc extends {
    writable: false
  } ? Readonly<InferValue<Prop, Desc>> :
  Desc extends {
    writable: true
  } ? InferValue<Prop, Desc> :
    Readonly<InferValue<Prop, Desc>>
```

First, we deal with the `writeable` property of a `PropertyDescriptor`. It's a set of conditions to define some edge cases and conditions of how the original property descriptors work:

1. If we set `writable` and any property accessor (`get`, `set`), we fail. `never` tells us that an error was thrown.
2. If we set `writable` to `false`, the property is read-only. We defer to the `InferValue` helper type.
3. If we set `writable` to `true`, the property is not read-only. We defer as well.
4. The last, default case is the same as `writeable: false`, so `Readonly<InferValue<Prop, Desc>>`. (`Readonly<T>` is built-in.)

This is the `InferValue` helper type, dealing with the set `value` property:

```
type InferValue<Prop extends PropertyKey, Desc> =
  Desc extends { get(): any, value: any } ? never :
  Desc extends { value: infer T } ? Record<Prop, T> :
  Desc extends { get(): infer T } ? Record<Prop, T> :
  never;
```

Again a set of conditions:

1. If we have a getter and a value set, `Object.defineProperty` throws an error, so `never`.
2. If we have set a value, let's infer the type of this value and create an object with our defined property key and the value type.
3. Or we infer the type from the return type of a getter.
4. Anything else we forgot. TypeScript won't let us work with the object as it's becoming `never`.

Moving It to the Object Constructor

This already works wonderfully in your code, but if you want to make use of that throughout the whole application, we should put this type declaration in `ObjectConstructor`. Let's move our helpers to *mylib.d.ts* and change the `ObjectConstructor` interface:

```
type InferValue<Prop extends PropertyKey, Desc> =
  Desc extends { get(): any, value: any } ? never :
  Desc extends { value: infer T } ? Record<Prop, T> :
  Desc extends { get(): infer T } ? Record<Prop, T> :
  never;

type DefineProperty<
  Prop extends PropertyKey,
  Desc extends PropertyDescriptor> =
    Desc extends {
      writable: any, set(val: any): any
    } ? never :
    Desc extends {
      writable: any, get(): any
    } ? never :
    Desc extends {
      writable: false
    } ? Readonly<InferValue<Prop, Desc>> :
    Desc extends {
      writable: true
    } ? InferValue<Prop, Desc> :
      Readonly<InferValue<Prop, Desc>>

interface ObjectConstructor {
  defineProperty<
    Obj extends object,
    Key extends PropertyKey,
    PDesc extends PropertyDescriptor
  >(obj: Obj, prop: Key, val: PDesc):
    asserts  obj is Obj & DefineProperty<Key, PDesc>;
}
```

Thanks to declaration merging and function overloading, we attach this much more concrete version of `defineProperty` to `Object`. In use, TypeScript aims for the most correct version when selecting an overload. So we always end up with the one overload where we bind generics through inference. Let's see what TypeScript does:

```
const storage = {
  currentValue: 0
}

Object.defineProperty(storage, 'maxValue', {
  writable: false, value: 9001
})

storage.maxValue // It's a number
storage.maxValue = 2 // Error! It's read-only

const storageName = 'My Storage'
defineProperty(storage, 'name', {
  get() {
    return storageName
  }
})

storage.name // It's a string!

// It's not possible to assign a value and a getter
Object.defineProperty(storage, 'broken', {
  get() {
    return storageName
```

```
  },
  value: 4000
})

// Storage is never because we have a malicious
// property descriptor
storage
```

We already have some great additions to regular typings that we can reuse in all our applications. Take care of this file and make it part of your standard setup.

Epilogue: Lesson 50

Welcome to the last lesson in this book. When we set out on this journey, we had a specific focus: how can we use TypeScript as an extension to JavaScript, the programming language that drives the web?

Following this route has led us deep into the realms of type systems, learning how we can model data with union and intersection types, model behavior with generics and conditional types, and trying to reduce type maintenance as much as possible.

By the end, we had an arsenal of tools at our disposal, helping make sure we use the least amount of code possible to define the most complex JavaScript scenarios.

Still, this is just part of what TypeScript has to offer. Arguably the most important part, but you never know where you'll end up with your newfound TypeScript skills.

In this final lesson, I want to prepare you for the unknown. Not the type `unknown`, but what lies ahead: the future, which can't be covered in a book that is supposed to be timeless.

So where do you go from here?

Listen

I urge you to lend an ear to the TypeScript team. They work in the open. You can see their team communication on GitHub,[30] find roadmaps, upcoming features, and their overall plans. We know that even if TypeScript already covers a wide range of JavaScript scenarios, there are still situations where we need a type cast, or worse: `any`! This is supposed to change. And the TypeScript team is very vocal about their plans. The roadmap[31] especially is a good read.

You should also keep an ear out for TC39, the committee that standardizes ECMAScript and that works closely with the TypeScript team. There's even a little overlap in membership. New language features are specified by TC39 and implemented in TypeScript once they reach a certain maturity level and are ready to be implemented in JavaScript engines. Their GitHub repo[32] is an excellent source of discussions and new features.

Learn

We covered a large and significant part of TypeScript, but there is more to it. There are features that go deep into

30 https://smashed.by/typescriptteam
31 https://smashed.by/typescriptroadmap
32 https://smashed.by/ecma262

object-oriented programming languages. Other language constructs just make certain forms easier to write. And there are experimental features that are required by certain frameworks. The official TypeScript handbook and documentation[33] are excellent sources for all this.

You will find starter kits for TypeScript with your favorite framework. You'll also get a good overview of language features that you haven't yet seen and might find useful. As you know from the interludes in this book, people can be very opinionated about those features. But try them for yourself.

Read On

The TypeScript community is very active in providing new tools, writing articles, and showing developers what can be done with the power of this language. There is a new Node.js-like runtime called Deno,[34] developed by Node.js's original creators, which supports TypeScript out of the box.

Together with the people from the package manager Pika[35] they make sure that you get type declaration files over HTTP once you import a Deno package from a URL.

33 https://typescriptlang.org
34 https://deno.land/
35 https://www.pika.dev/

Then there's a myriad of blogs on TypeScript.

Marius Schulz[36] has been writing for years about TypeScript and curates a wonderful newsletter called TypeScript Weekly.[37] Sometimes you find articles in there by me, which you can read on my website.[38]

As the hard rock poets Deep Purple so masterfully said: listen, learn, read on. There's a lot to uncover, a lot to learn!

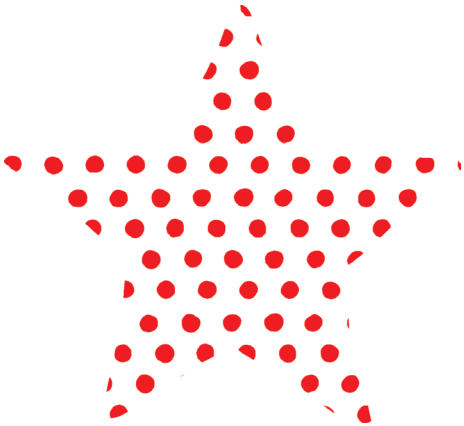
Thank you for reading my book!

I hope you had as much joy reading it as I had writing it. TypeScript is always evolving, and I always try to find new and exciting solutions to typing challenges. Get in touch with me on Twitter at @ddprrt,[39] and let's figure out how we can type your code best!

36 https://mariusschulz.com/
37 https://www.typescript-weekly.com/
38 https://fettblog.eu
39 https://smashed.by/ddprrt

THE LESSONS

Index



More Smashing Books

We pour our heart and soul into crafting books that help make the web better. We hope you'll find these other books we've published useful as well—and thank you so much for your kind support from the very bottom of our hearts.

- *Click! How to Encourage Clicks Without Shady Tricks* by Paul Boag
- *The Ethical Design Handbook* by Trine Falbe, Martin Michael Frederiksen and Kim Andersen
- *Inclusive Components* by Heydon Pickering
- *Art Direction for the Web* by Andy Clarke
- *Form Design Patterns* by Adam Silver
- *Design Systems* by Alla Kholmatova
- *Smashing Book 6: New Frontiers in Web Design* by Laura Elizabeth, Marcy Sutton, Rachel Andrew, Mike Riethmuller, Lyza Gardner, Yoav Weiss, Adrian Zumbrunnen, Greg Nudelman, Ada Rose Cannon, & Vitaly Friedman


The world is a miracle. So are you.
Thanks for being smashing.
SMASHING
MAGAZINE

"This is a gentle and timeless journey through the tenets of TypeScript. If you're a JavaScript programmer looking for a clear primer text to help you become immediately productive with TypeScript, this is the book you're looking for. It's filled with practical examples and clear technical explanations."

—***Natalie Marleny***, *Application Engineer*

"Stefan walks you through everything from basic types to advanced concepts like the infer keyword in a clear and easy to understand way. The book is packed with many real-world examples and great tips, transforming you into a TypeScript expert by the end of it. Highly recommended read!"

—***Marvin Hagemeister***, *Creator of Preact-Devtools*

Stefan Baumgartner is a software architect based in Austria. He has published online since the late 1990s, writing for Manning, Smashing Magazine, and A List Apart. He organizes ScriptConf, TSConf:EU, and DevOne in Linz, and co-hosts the German-language Working Draft podcast.

SMASHING MAGAZINE

9 783945 749906