

TYPESCRIPT



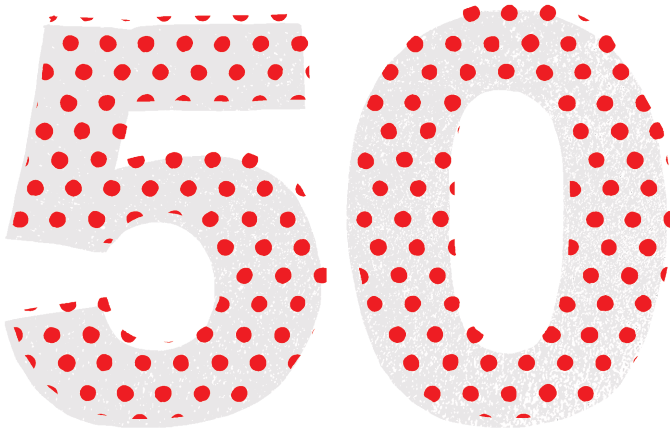
50

LESSONS

by

STEFAN BAUMGARTNER

TYPESCRIPT



LESSONS

by

STEFAN BAUMGARTNER

Published 2020 by Smashing Media AG, Freiburg, Germany.
All rights reserved.
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

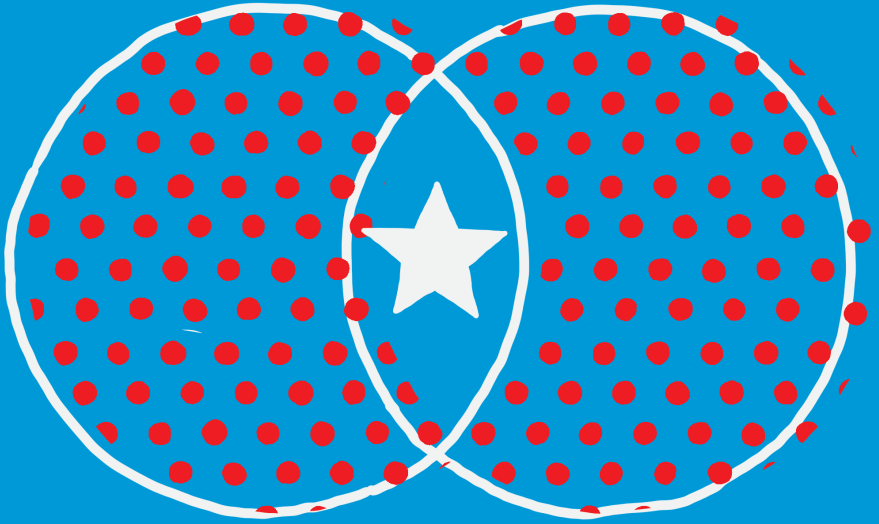


To Doris, Clemens, and Aaron

Table of Contents

	<i>Introduction</i>	<i>xi</i>
1	TypeScript for Smashing People	19
2	Working with Types	67
3	Typing Functions	131
4	Union and Intersection Types	201
5	Generics	267
6	Conditional Types	329
7	Thinking in Types	383





Chapter Four



UNION AND INTERSECTION TYPES

Lesson 22: Modeling Data	204
Lesson 23: Moving in the Type Space	212
Lesson 24: Working With Value Types	221
Lesson 25: Dynamic Unions	232
Lesson 26: Object Types and Type Predicates	239
Lesson 27: Down at the Bottom: never	246
Lesson 28: Undefined and Null	253

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,  
  url: string,  
  talks: Talk[]  
}
```

```
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

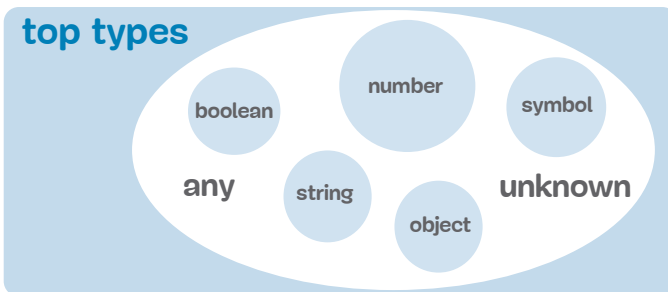
²⁴ <https://smashed.by/typingintro>



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.

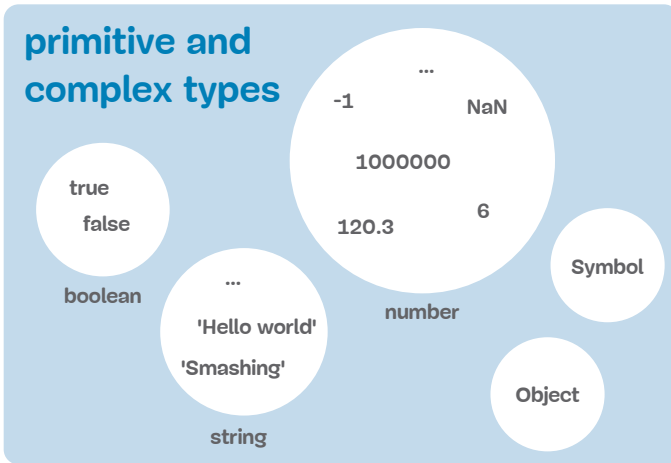
The part we want to focus on is the “set of allowed values.” This is something we already experienced when working with types. Once a variable has a certain type annotation, TypeScript only allows a specific set of values to be assigned.

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.



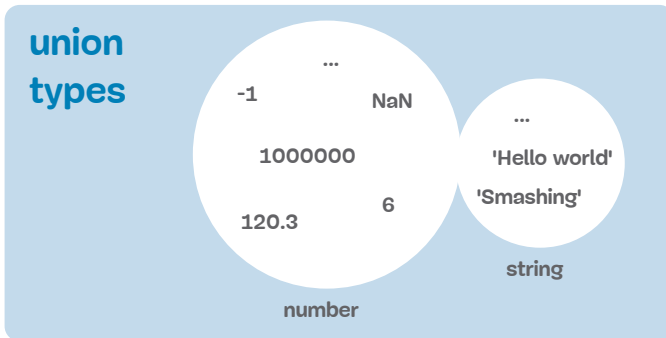
Top types, including all other types.

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',
```

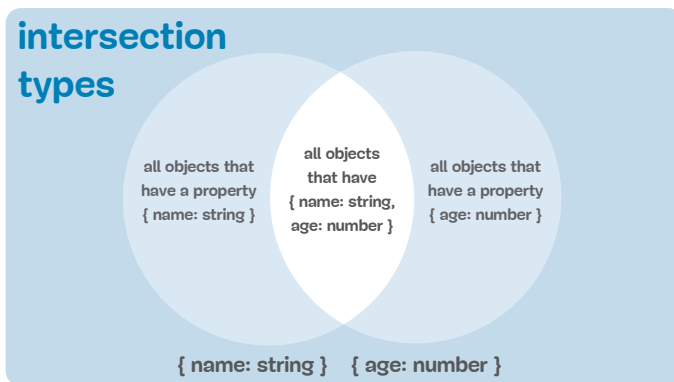
```
    age: 38
  }
```

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`.

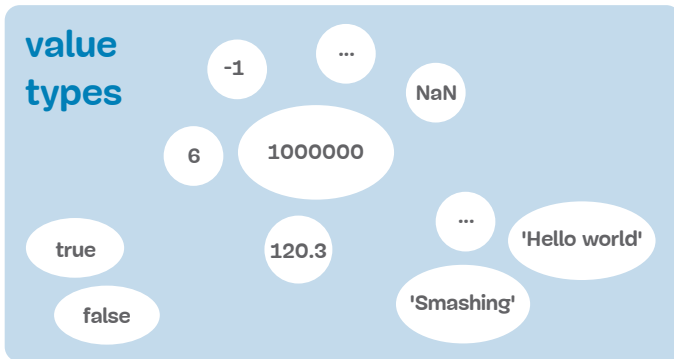
With that, only the variable `me` becomes compatible with the newly generated type. The intersection is a subset of both `Age` and `Name` sets – smaller, narrower, and we have to be more explicit about our values.

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.

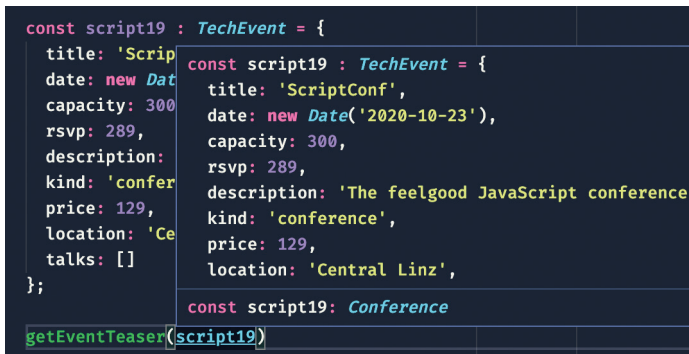
This means the property `kind` in `script19` will not be inferred as `'conference'` but as `string`. And `string` is a much wider set of values than `'conference'`. For this to work, we need to tell TypeScript again that we are looking for the value type, not for its superset of types. We have a couple 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`.



```
const script19 : TechEvent = {
  title: 'ScriptConf',
  date: new Date('2020-10-23'),
  capacity: 300,
  rsvp: 289,
  description: 'The feelgood JavaScript conference',
  kind: 'conference',
  price: 129,
  location: 'Central Linz',
  talks: []
};

getEventTeaser(script19)
```

The screenshot shows a code editor with a variable `script19` of type `TechEvent`. The variable is assigned an object with properties: `title` ('ScriptConf'), `date` (a `Date` object for '2020-10-23'), `capacity` (300), `rsvp` (289), `description` ('The feelgood JavaScript conference'), `kind` ('conference'), `price` (129), `location` ('Central Linz'), and `talks` (an empty array). A function call `getEventTeaser(script19)` is shown below. A tooltip is displayed over `script19`, showing the inferred type `Conference` and the corresponding object structure.

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.

```
const script19 = {
  title: 'Script conference',
  date: new Date('2019-10-25'),
  capacity: 300,
  rsvp: 289,
  description: 'The feelgood JS conference',
  kind: 'conference' as const,
  price: 129,
  location: 'Central Linz',
  talks: [{
    speaker: 'Vitaly Friedman',
    title: 'Designing with Privacy in mind',
    abstract: '...'
  }]
};
```

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(e1 => e1.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(e1 => {
    grouped[e1.kind].push(e1)
  })
  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[],  
  rsvp: 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 `${event.title} (Webinar)`
    case 'hackathon':
      return `${event.title} (Hackathon)`
    default:
      (parameter) event: never
      event
      throw new Error('No idea what to do')
  }
}
```

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 squiggle 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`.

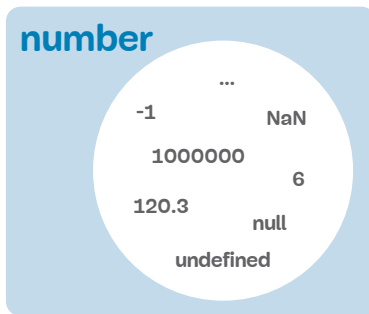
Both `null` and `undefined` denote the absence of a value. `undefined` tells us that a variable or property has been declared, but no value has been assigned. `null`, on the other hand, is an *empty value* that can be assigned to clear a variable or property.

Both values are known as *bottom values*, values that have no actual value.

Douglas Crockford once said²⁵ 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`. Programmatically, 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 `

## ${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)
}
```

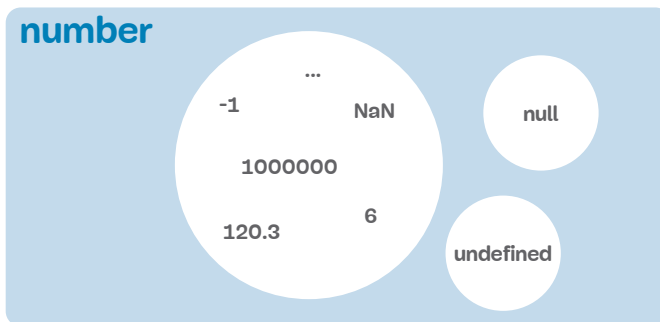
And here's the problem: we have to be very sure that an element with the ID `event-list` exists in our HTML. Otherwise `document.querySelector` returns `null`, and appending the list will break the application.

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 chaining²⁶), 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 operator²⁷ 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.



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.

“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.

