TypeScript for React Devs

Michel Weststrate - @mweststrate - https://michel.codes

https://ralondon2019.surge.sh

Wifi: etcvenues / Wifi7109

Pre-requisites: `yarn`, recent version of node, `git`, VSCode

Clone and `yarn install`: https://github.com/mweststrate/react-ts-conversion-exercise

About Me

@mweststrate - https://michel.codes

4 years full-time TypeScript experience

300 KLoc code base @Mendix

Open source projects: MobX, Mobx-state-tree, etc

Introduce yourselves!

- Experience with
  - JavaScript
  - TypeScript
  - Other static languages (Java / C#)

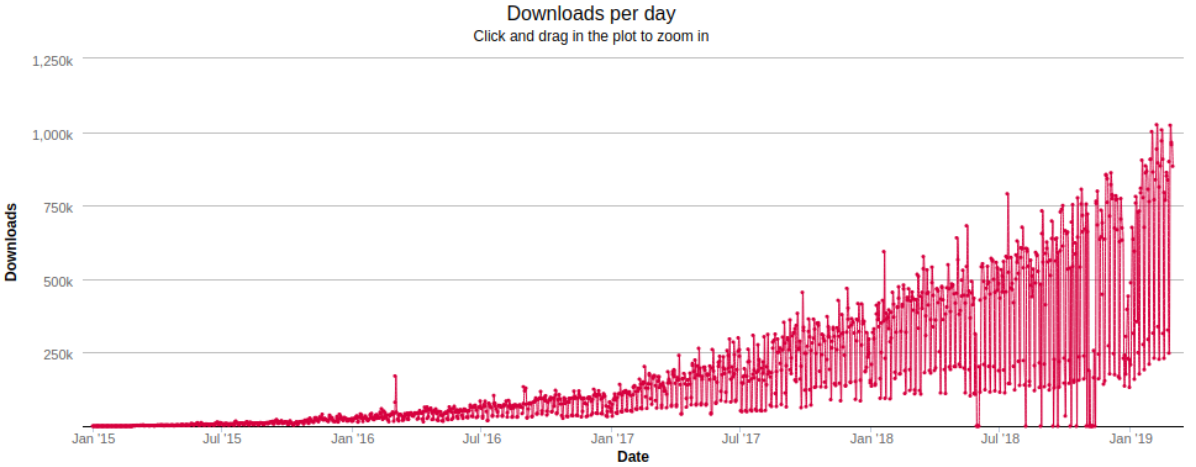
Symbols

? Question

✔ Answer

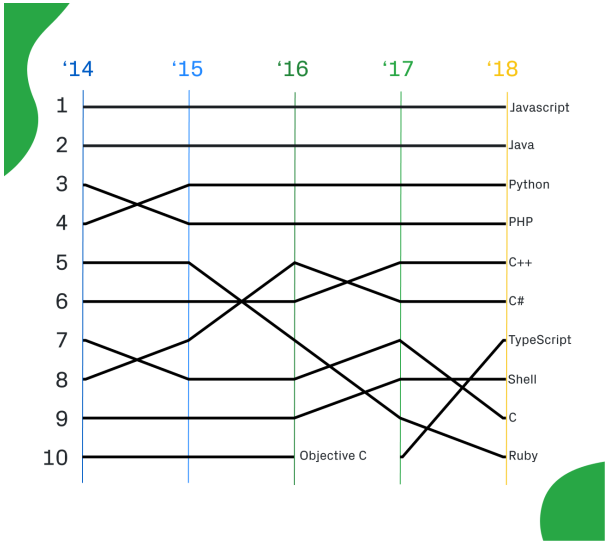
Detail

Adoption



Microsoft, Google, Facebook(!)

Adoption



https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/

Adoption

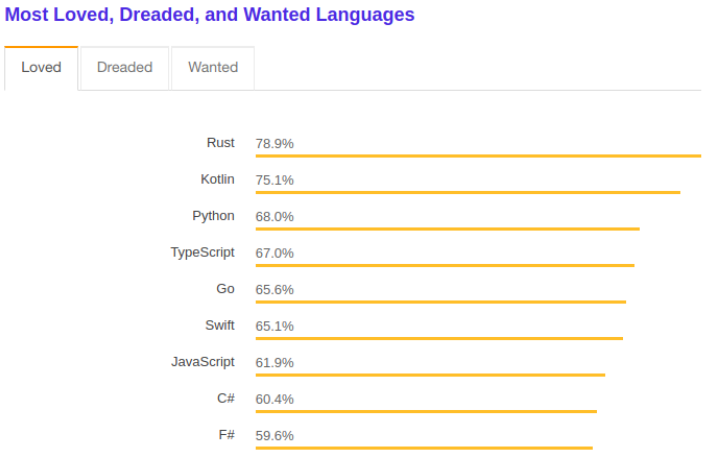
Brian Holt Retweeted

David K. 🎹 @DavidKPiano · 18h

The five stages of @typescriptlang:

😉 Denial: "This doesn't make things any easier"
😠 Anger: "Missing index signature!?"
🤔 Bargaining: "<any>"
😞 Depression: "I have to refactor everything"
🙂 Acceptance: "Never doing another JS project without TS ever again"

Adoption



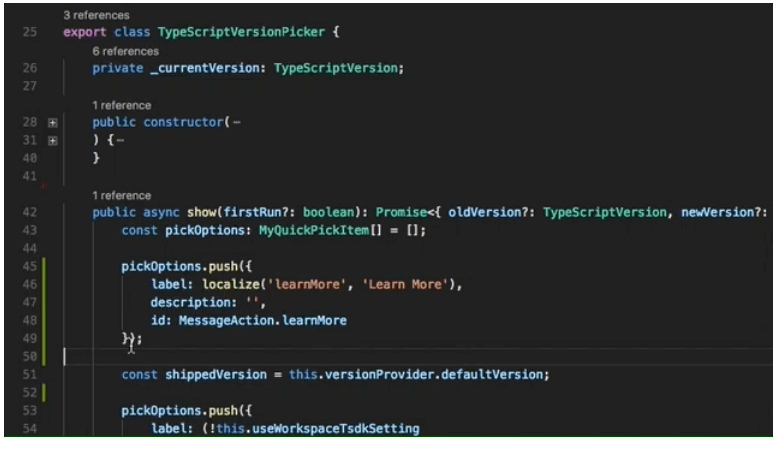
https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted

Why TypeScript?

4 reasons

1. Catches more bugs?

2. Maintenance / refactoring speed



3. Discoverability

- Code becomes self documenting
- Types are a good trigger to think design first

4. Type-first leads to more consistent API design

michel.codes
@mweststrate

I love starting with @typescriptlang in projects, for the simple reason it usually helps to design better api's. something hard to type? Then probably hard to learn. Not always true, but a good rule of 👍.

below is a nice example how thinking in types earlier could have helped

Ryan Cavanaugh @SeaRyanC
Have you ever really looked at some of the library code we use every day (or used to)?

Let's look at underscore's "sample" method, which produces some amazing…
Deze collectie tonen

Tweet vertalen

23:22 - 23 jan. 2019

16 retweets **84** vind-ik-leuks

Thread

5. Bonus

TypeScript transpiles features to ES3 / ES5

- Arrow functions
- Classes
- Decorators
- `async / await`
- Object spread
- Iterators
- ESM / CommonJS modules
- ...all standardized ES6 language features

Awesome! Let's migrate all the things

Well...

- Make sure you have one experienced TS dev on the team
- Make sure all team members are on board
- Spending hours on fixing compiler errors on perfectly working code is extremely demotivating (less strict settings will help)
- Pick your battles (super generic utilities are typically hard to type)

TypeScript principles

- A superset: all valid JavaScript is valid TypeScript (syntactically)
- Be able to type common, *existing* JS patterns

Two types of compile errors

1. Type errors
2. Unrecoverable errors

TypeScript limitations

No run-time typechecking

```
async function getUserProfile(): Promise<UserProfile> {
    const response = await window.fetch("/profile")
    return response.json()
}

const user = await getUserProfile()
console.log(user.address.country)
```

TypeScript disappears at runtime

```
// mobx
function set(index: number | string, value: any): void {
    this.data[index] = value
}
```

TypeScript disappears at runtime

```
function set(index: number | string, value: any): void {
    if (typeof index !== "number" && typeof index !== "string")
        throw "Expected string or number as index"
    this.data[index] = value
}
```

TypeScript disappears at runtime

```
function set(index: number | string, value: any): void {
    if (
        process.env.NODE_ENV !== "production" &&
        (typeof index !== "number" && typeof index !== "string")
    ) {
        throw "Expected string or number as index"
    }
    this.data[index] = value
}
```

So, don't use static type checking?

"I stopped using using the Word spelling checker, because it didn't correct: "
At what time shall we meat?

Getting started

Run TS on any code base

`// @tscheck`

Picks up JS Doc comments

TypeScript allows mixing `.js` and `.ts` files

DEMO

Use JSDocs as types

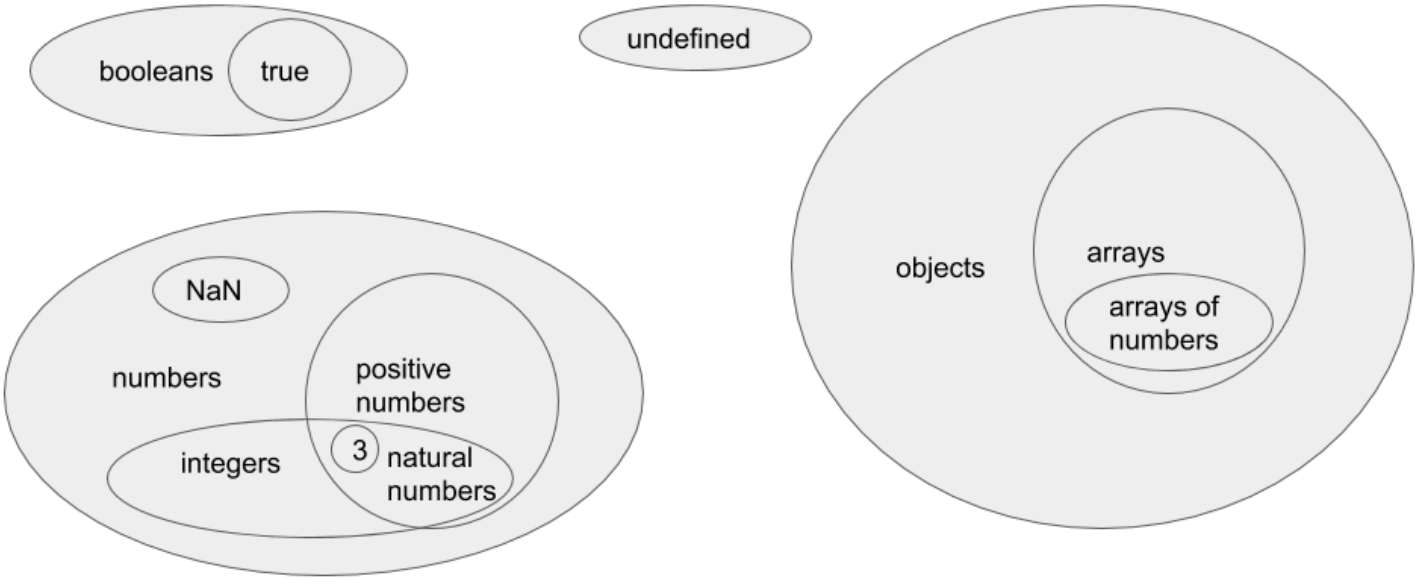
Setting up a project

```
yarn add -D typescript

yarn tsc --init

yarn add -D @types/libraryXYZ
```

Let's get started with types!

What are types?

all possible values in the world



Primitives and flow analysis

All examples today are based on the most strict compiler settings

Type annotations

Typically, behind `":"`

```
let timeout: number;

timeout = 3 // OK
timeout = "Test" // Error
```

Built-in primitives

- `string` (lowercase!)
- `number`
- `boolean`
- `null`
- `undefined`

Control flow analysis

```
let timeout: number

if (theProcessIsAsync()) {
  timeout = 3
}

console.log("Got timeout: " + timeout)
```

? This will error, why?

Control flow analysis

Fix 1: Ensure `timeout` is a `number`:

```
let timeout: number

if (theProcessIsAsync()) {
  timeout = 3
} else {
  timeout = 0
}

console.log("Got timeout: " + timeout)
```

Control flow analysis

Fix 1: Ensure `timeout` is a `number`:

```
let timeout: number = 0

if (theProcessIsAsync()) {
  timeout = 3
}

console.log("Got timeout: " + timeout)
```

Control flow analysis

Fix 2: Allow the `timeout` to be `undefined`:

```
let timeout: number | undefined

if (theProcessIsAsync()) {
  timeout = 3
}

console.log("Got timeout: " + timeout)
```

Control flow analysis

Fix 2: Allow the `timeout` to be `undefined`:

```
let timeout: number | undefined

if (theProcessIsAsync()) {
  timeout = 3
}

console.log("Got timeout: " + timeout)
if (timeout =/= undefined) {
  console.log("In milliseconds, that is: " + timeout * 1000)
}
```

The union "|" operator and `undefined` type

```
let timeout: number | undefined
```

The type of `3` is `number`.

Likewise, the JavaScript value `undefined` has the TypeScript type `undefined`.

`thing: A | B` can be read as *thing* is of type *A* or *B*.

Quiz

```
let firstName = "michel"
const lastName  = "weststrate"
```

? What are the inferred types?

The `type` keyword

The `type` keyword

Types are handy aliases for more complex types.

The `type` keyword

```
let timeout: number | undefined
let port: number | undefined
```

```
type MaybeNumber = number | undefined

let timeout: MaybeNumber
let port: MaybeNumber
```

Functions

Functions

```
function secondsToMilliseconds(seconds: number): number {
  return seconds * 1000
}
```

Functions types

```
type NumberToNumber = (value: number) ⇒ number

const secondsToMilliseconds: NumberToNumber = seconds ⇒ seconds * 1000
```

Optional arguments

```
function printError(msg: string, error?: Error) {
  console.error(msg, error)
}
```

Default arguments

```
function printError(msg: string = "Something bad happened!", error?: Error) {
  console.error(msg, error)
}

printError() // Something bad happened!
```

Function overloading

```
printError(msg)
printError(msg, Error)
printError(Error)
```

Function overloading

```
function printError(msg: string)
function printError(msg: string, error: Error)
function printError(error: Error)
function printError(msgOrError: string | Error, maybeError?: Error) {
```

Beware of function overloading

Variadic arguments

```
function printError(msg: string)
function printError(msg1: string, msg2: string)
function printError(msg1: string, msg2: string, msg3: string) // etc etc
function printError() {
  for (let i = 0; i < arguments.length; i++) console.log(`Message ${i}: ${arguments[i]}`)
}
```

Arrays

Arrays

```
const x = [1, 2, 3]

// inferred type:
const x: number[]

// convenience syntax for
const x: Array<number>
//               ^ Generic argument!
```

ReadOnlyArray

`ReadOnlyArray<T>`

Recommended type to use when returning arrays from methods; makes clear they are not to be modified!

Mixed types arrays

```
const x: (number | string)[] = ["hello", "world", 42]
```

Tuples

```
function Counter() {
  const [count, setCount] = useState(0)
//         ^ integer
//                   ^ function
//                               ^ ???

  return <div>
    {count}
    <button onClick={() => setCount(count => count + 1)}>
      increment
    </button>
  </div>
}
```

Tuples

```
function useState(initial: number): [number, Function]
```

- Arrays: dynamic length arrays, consistent type: `X[]`
- Tuples: "fixed" length arrays, mixed types (but orderly) `[X, Y, ...Z]`

Collection quiz: arrays

```
function printFirstValue(numbers: number[]) {
  const first = numbers[0]
  console.log(first)
}
```

Collection quiz: maps

```
function printFirstValue(numbers: Map<string, number>) {
  const first = numbers.get("x")
  console.log(first)
}
```

TypeScript is a pragmatic language

Non-null assertion operator

```
function printFirstValue(numbers: Map<string, number>) {
  const first = numbers.get("x")! // ← note the exclamation mark!
  console.log(first)
}
```

`first` is now of type `number`

Non-null assertion operator

Optional Chaining to the rescue

```
const x = { a: undefined }
console.log(x.a?.b?.c) // undefined
```

Optional Chaining: Function calls

```
async function makeRequest(url: string, log?: (msg: string) => void) {
    log?.(`Request started at ${Date.now()}`);
    // etc...
}
```

Nullish Coalescing

```
const localStorage: { volume: number } = getDataFromSomewhere()

function initializeAudio() {
  let volume = localStorage.volume || 0.5
}
```

Object types

Object types

```
type Todo = {
  title: string
  done: boolean
  description?: string
}
```

```
const todo: Todo = {
  title: "Test",
  done: false
}
```

Freshness

```
type Todo = {
  title: string
  done: boolean
  description?: string
}

const todo: Todo = {
  title: "Test",
  done: false,
  test: 123
}
```

? Does the above result in a compile error?

Freshness

```
type TextAreaProps = {
  caption: string
  onChange: (newValue: string) ⇒ void
  writable?: boolean
}

class TextArea extends React.Component<TextAreaProps> {
  // ....
}

// Later ...
;<TextArea caption="Name" onChange={/* .. */} writeable={true} />
//                                    ^ caught!
```

Index signatures

```
type Todo = {
  title: string
  done: boolean
  description?: string
}

const todosById = {}

todosById["x23c-241b"] = { title: "Grok index signature" }
```

? What are the two problems with in this statement?

Index signatures

```
type Todo = {
  /* */
}

const todosById: {
  [key: string]: Todo
} = {}

todosById["x23c-241b"] = {
  title: "Grok index signature"
}
```

1. `[ .. ]` - there are dynamic properties
2. `[key: string]` - where the property name is any valid `string`
3. `[...]: Todo` - and the value is a `Todo`

Interfaces vs types

```
type Todo = {
  title: string
  done: boolean
  description?: string
}

interface Todo {
  title: string
  done: boolean
  description?: string
}
```

Interfaces vs types

```
type Todo = {
  title: string
  done: boolean
  description?: string
  subtasks: Todo[] // Compile error
}

interface Todo {
  title: string
  done: boolean
  description?: string
  subtasks: Todo[] // OK!
}
```

Interfaces vs types

When to use which?

- 95% of times it doesn't matter
- Establish team preference

My preferences:

- For function "options" -> `type`
- For data models / network payloads etc -> `interface`
- Standardized API's (`Loggable`, `Serializable`) -> `interface`
- Like tabs vs. spaces...: `¯\_(ツ)_/¯`

Exercises part 1

Open and fork https://stackblitz.com/edit/typescript-8xntyp?file=index.ts

Go to the folder `exercises/part1`.

Make sure that for each exercise in the folder the type errors are fixed. Positive tests should pass, negative tests should cause compile errors.

Tips:

- consider pair programming!
- you can use your favorite IDE

Solutions

spoiler!

Exercise 1

```
const x: (number | string)[] = ["hello", "world", 42]
```

Exercise 2

```
type Todo = {
  title: string
  done: boolean
  description?: string
  [key: string]: any
}
```

Exercise 3

Multiple solutions:

```
function concatMessages(message1: string, ...rest: string[]) {
  return [message1].concat(rest).join(", ");
}
```

```
function concatMessages(message: string, ...rest: string[]): string;
function concatMessages(...messages: string[]) {
  return messages.join(", ");
}
```

```
function concatMessages(...messages: [string, ...string[]]) {
  return messages.join(", ");
}
```

Combining object types

Google Documenten


Documenten


Spreadsheets


Presentaties

Extends

```
interface Shareable {
  sharedWith: string[]
}

interface Previewable {
  thumbnail: string
}

interface Document extends Shareable, Previewable {
  contents: string
}
```

Intersection

```
type Shareable = {
  sharedWith: string[]
}

type Previewable = {
  thumbnail: string
}

type Document = Shareable & Previewable & {
  contents: string
}
```

Intersection

```
type A = { a: number, b: number }
type B = { b: string, c: string }
type C = A & B
```

? What is the type of c ?

Intersection - real life example:

```
import * as express from "express"
import { auth } from "express-openid-connect"

const app = express()
const port = 3000

app.use('/', auth())
app.get('/', (req: IncomingMessage & { openid: { user: { name: string }}}, res) => {
  res.send(`hello ${req.openid.user.name}`);
})
```

Union

```
type StoplightColor = "red" | "green" | "yellow"

type Asset = Spreadsheet | Presentation | Document
```

Discriminated Unions

```
interface Spreadsheet {
  contents: Column[]
}
interface Presentation {
  slides: Slide[]
}
interface Document {
  contents: string
}
type Asset = Spreadsheet | Presentation | Document

function printContents(asset: Asset) {
  if ((asset as any).slides) {
    (asset as Presentation).slides.forEach(slide => console.log(slide.contents))
  }
  else if (typeof (asset as any).contents === "string") {
    console.log((asset as Document).contents)
  }
  else if (Array.isArray((asset as any).contents)) {
    (asset as Spreadsheet).contents.forEach(column => console.log(column.name))
  }
  else {
    throw new Error("Weird document")
  }
}
```

```
 interface Spreadsheet {
  kind: "spreadsheet"
  contents: Column[]
}
interface Presentation {
  kind: "presentation"
  slides: Slide[]
}
interface Document {
  kind: "document"
  contents: string
}
type Asset = Spreadsheet | Presentation | Document

function printContents(asset: Asset) {
  if (asset.kind === "presentation") {
    asset.slides.forEach(slide => console.log(slide.contents))
  }
  else if (asset.kind === "document") {
    console.log(asset.contents)
  }
  else if (asset.kind === "spreadsheet") {
    asset.contents.forEach(column => console.log(column.name))
  }
  else {
    throw new Error("Weird document")
  }
```

"So yeah, I'll tell the backend guys to introduce a `kind` field to all their output so I can make TS happy..."

```
interface Spreadsheet {
  contents: Column[]
}
interface Presentation {
  slides: Slide[]
}
interface Document {
  contents: string
}
type Asset = Spreadsheet | Presentation | Document

function isDocument(asset: any): asset is Document {
  return asset && typeof asset.contents === "string"
}

function isSpreadsheet(asset: any): asset is Spreadsheet {
  return asset && Array.isArray(asset.contents)
}

function isPresentation(asset: any): asset is Presentation {
  return asset && Array.isArray(asset.slides)
}

function printContents(asset: Asset) {
  if (isPresentation(asset)) {
    asset.slides.forEach(slide => console.log(slide.contents))
  }
  else if (isDocument(asset)) {
```

Type Guards

```
function isDocument(asset: any): asset is Document {
  return asset && typeof asset.contents === "string"
}

function isSpreadsheet(asset: any): asset is Spreadsheet {
  return asset && Array.isArray(asset.contents)
}

function isPresentation(asset: any): asset is Presentation {
  return asset && Array.isArray(asset.slides)
}
```

Return type: `<parameter> is <TypeOrGeneric>`

Classes

Classes

```
abstract class DocumentBase implements Asse
  public readonly creationDate = new Date
  public author?: string

  constructor() {
    this.resetToDefault()
  }

  protected save() { /* code */  }

  abstract resetToDefault(): void
}
```

1. Field assignment
2. `abstract` keyword (class and method)
3. `readonly` keyword
4. `public`, `protected`, `private`
5. `?` nullable modifier
6. `extends`, `implements`

Classes

💡 Make a habit of using `readonly` by default!

💡 `public`, `protected` and `private` only have compile time meaning. Not to be confused with private fields proposal.

💡 One might not need `extends`....

Quiz

```
class Document {
  contents: string
}
```

Class with type assertion

```
abstract class DocumentBase {
  constructor() {
    this.resetToDefault()
  }

  abstract resetToDefault(): void
  /* more */
}

class Document extends DocumentBase {
  private contents!: string // Note the non-null assertion!

  constructor(contents?: string) {
    super()
    if (contents) { this.contents = contents }
  }

  save() {
    this.contents = this.contents.trim()
    super.save()
  }

  public resetToDefault() {
    this.contents = ""
  }
}
```

Constructor fields

```
class Point {
  public x: number
  public y: number

  constructor(x: number, y: number) {
    this.x = x
    this.y = y
  }
}
```

Generics

Generics

```
function debug(value: ????): ???? {
  console.log(value)
  return value
}

if (debug(user).isLoggedIn) {
  //            ^ strongly typed!
}
```

Generics

Generic type: a type that receives another type as argument.

```
const numbers: Array<number> = [1, 2, 3]

const odd = numbers.filter(n => n % 2 === 1)
// Filter produces: Array<number>

const strings: Array<string> = ["My", "Name", "Tarzan"]

const longWords = strings.filter(s => s.length > 4)
// Filter produces: Array<string>
```

Generics

```
type Todo = {
  title: string
  done: boolean
}

const todos = new Map<string, Todo>()

todos.set("a17", {
  title: "Learn Generics",
  done: true
})

todos.get("b23") // Todo | undefined
```

Generics

```
interface Map<K, V> {
  readonly size: number
  has(key: K): boolean
  get(key: K): V | undefined
  set(key: K, value: V): this
  clear(): void
  delete(key: K): boolean
  forEach(callbackfn: (value: V, key: K, map: Map<K, V>) => void, thisArg?: any): void
}
```

Default generic arguments

```
class Map<KEY_TYPE = string, VALUE_TYPE = any> {
  //
}

const valueStore = new Map<number>()
```

Generics and functions

```
function first<ITEM_TYPE>(thing: ITEM_TYPE[]): ITEM_TYPE | undefined
function first<ITEM_TYPE>(thing: Map<any, ITEM_TYPE>): ITEM_TYPE | undefined
function first(thing: unknown): any {
  if (Array.isArray(thing)) return thing[0]
  else if (thing instanceof Map && thing.size > 0) return thing.values().next().value
  return undefined
}

const a = first(["Noa", "Veria"]) // string
const b = first(new Map([["Noa", 7], ["Veria", 5]])) // number
```

Generic constraints using `extends`

```
function assertNonEmpty<TYPE>(thing: TYPE): TYPE {
  if (thing.length === 0) throw new Error("Assertion error: empty")
  return thing
}

const numbers = [1, 2, 3]
assertNonEmpty(numbers).reduce(/* ... */)

const aString = "Test"
console.log("Tail: " + assertNonEmpty(aString).substr(1))
```

Things that can be generic

```
interface IMap<T> {
  get(key: string): T
}

class Map<T> implements IMap<T> {
  get(key: string): T
}

function first<T>(map: IMap<T>): T {
  // something
}

type IMap2<T> = {
  get(key: string): T
}

type Predicate<T> = (value: T) ⇒ boolean
```

Exercises part 2

Open and fork https://stackblitz.com/edit/typescript-8xntyp?file=index.ts

Go to the folder `exercises/part2`.

Make sure that for each exercise in the folder the type errors are fixed. Positive tests should pass, negative tests should cause compile errors.

Solutions

spoiler!

Exercise 1

```
class MyCollection<T> {
  private data: T[] = []

  constructor(initial?: Iterable<T> | ArrayLike<T>) {
    if (initial) this.data = Array.from(initial)
  }

  get(index: number): T {
    return this.data[index]
  }

  add(value: T): void {
    this.data.push(value)
  }

  forEach(callback: (item: T, index: number) => void): void {
    for (let i = 0; i < this.data.length; i++) {
      callback(this.data[i], i)
    }
  }

  map<O>(callback: (item: T, index: number) => O): O[] {
    return this.data.map(callback)
  }
}
```

Exercise 2

```ts
type Initializer<T> = () => T
type Updater<T> = (current: T) => T

function useState<T>(initial: Initializer<T> | T): [T, (next: Updater<T> | T) => void] {
  throw new Error("Not implemented") // ignore this line
}
```

Exercise 3

```
function assign<A, B>(object1: A, object2: B): A & B {
  throw "Not implemented" // ignore this line
}
```

Exercise 4

```
function assign<A, B, C, D, E>(obj1: A, obj2: B, obj3: C, obj4: D, obj5: E): A & B & C & D &
function assign<A, B, C, D>(obj1: A, obj2: B, obj3: C, obj4: D): A & B & C & D
function assign<A, B, C>(obj1: A, obj2: B, obj3: C): A & B & C
function assign<A, B>(obj1: A, obj2: B): A & B
function assign( ...objects: any[]): any {
  throw "Not implemented" // ignore this line
}
```

Better:

Typing modules

Using external modules

```
yarn add cool-package
```

4 possibilities

Using external modules

```
yarn add cool-package
```

😄 Module ships with own types! Hooray. Nothing to do!

Using external modules

```
yarn add cool-package
```

🙂 Doesn't ship with, but public types are available

Using external modules

```
yarn add cool-package
```

😒 No public types, use the package untyped

Using external modules

```
yarn add cool-package
```

😟No public types, include your own typings

How to create & publish a module that is consumable from JS and TS?

d.ts files

- "header" files, like `.h` for C.
- Public type signatures, no implementation
- Three things that can be declared:
  - Module exports
  - Global definitions
  - Ambient module declarations

d.ts files - module exports

```ts
// add.ts
export function add(a: number, b: number):number {
  return a + b
}
```

Compile with `--declaration --target es5 --module commonjs`

```ts
// add.js (generated)
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
function add(a, b) {
  return a + b
}
exports.add = add
```

```ts
// add.d.ts (generated)
export declare function add(a: number, b: number): number;
```

d.ts files - module exports

Consumption:

```
// resolves to add.js, using typings from add.d.ts
import { add } from "add"
```

d.ts files - shipping package with module types

- Enable `declaration` compiler option
- Make sure both `.js` and `.d.ts` files are packaged
- Add `types` field to `package.json`, which is the path to the types matching `main` field's `.js` file

```
{
  "name": "add",
  "main": "lib/add.js",
  "types": "lib/add.d.ts"
}
```

Project configuration / important compiler options

target

```
"target": "es6", /* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2
```

- Modern browsers only? `-> es6`
- All browsers? `-> es5`
- `TypeScript -> (rollup / webpack) -> Babel` stack? `-> es6`

module

```
"module": "commonjs", /* Specify module code generation: 'none', 'commonjs', 'amd', 'system',
```

- Node package? `-> commonjs`
- Browser? `-> ESNext` + rollup, webpack, etc

allowJS

```
"allowJs": true, /* Allow javascript files to be compiled. */
```

Enable during project migration

jsx

```
"jsx": "react",/* Specify JSX code generation: 'preserve', 'react-native', or 'react'. */
```

Enable in React (like) projects.

`.tsx` file extension *must* be used to be able to use jsx!

strict

```
"strict": true, /* Enable all strict type-checking options. */
```

- Enable strict for new projects
- In project migrations, it is often pretty hard if the `undefined` case is supposed to happen, best not enable it initially.

React and TypeScript

TL,DR:

Function components

```tsx
const Message = ({ message }: { message: string }) => <div>{message}</div>
```

Function components: statics

```
type MessageProps = { message: string }

const Message: React.FunctionComponent<MessageProps> = ({ message }) ⇒ <div>{message}</div>
```

Benefits:

- `Message.displayName`, `Message.propTypes`, `Message.defaultProps` will be know known
- `props.children` will be available

Class components

```
class Message extends React.Component<MessageProps, { count: number }> {
  static defaultProps = {
    message: "Hello world"
  }

  constructor(props: MessageProps) {
    super(props) // required in TS!
    this.state = { count: 0 }
  }

  render() {
    return (
      <div onClick={this.handleClick}>
        {this.props.message}
        clicked: {this.state.count}
      </div>
    )
  }
  // v --- often tricky to figure out
  handleClick = (e: React.MouseEvent<HTMLDivElement>) => {
    this.setState(state => ({
      count: state.count + 1
    }))
  }
}
```

PropTypes

? Still needed?

Important types

- `React.Component<PROPS, STATE>` and `React.PureComponent<PROPS, STATE>`: base class for components
- `React.FunctionComponent<PROPS>` (or `React.FC`): type for function components
- `React.ReactElement | null`: what is returned from `render`
- `React.ReactNode`: A tree of component instances (used for `children`)
- `React.MouseEvent<HTMLDivElement>` (etc): typing events. Note: `HTMLDivElement` is a DOM element, not a React element! Same applies to e.g. `RefObject<T>`
- `React.MouseEventHandler<HTMLDivElement>` typing event handlers

Tip

Many advanced React + TypeScript patterns (HoCs and such) are covered in: https://github.com/sw-yx/react-typescript-cheatsheet

Exercises part 3

- Exercise 1: Type a reducer correctly
- Exercise 2: Convert a React project

Exercise 1: Type a reducer correctly

Open and fork https://stackblitz.com/edit/typescript-8xntyp?file=index.ts

Follow the instructions in `exercises/part3/reducer.ts`

Exercise 2: Convert React project

- In this exercise we will convert a quite random project from the internetz to TypeScript.
- VSCode is recommended for this exercise.
- Clone the Parcel based React project from: https://github.com/mweststrate/react-ts-conversion-exercise
- Run `yarn install` and verify that`yarn start` results in a working application .
- TypeScript can be used with and without Babel. The Parcel bundler supports TypeScript out of the box, but uses Babel under the hood. So the build chain will: `TS -> TypeChecking -> Babel -> ES5 -> Parcel -> Bundling`
- Our first step is to make sure TypeScript is being picked up by the bundler. Install TypeScript, run `yarn add -D typescript` and then `yarn tsc --init` (or `node_modules/.bin/tsc --init`). The last command generates a compiler configuration for you, called `tsconfig.json`
- To make the initial migration easy, make sure the following compiler options are set: `"allowJS": true`, `"jsx": "react"`, `"strict": false`, `"noEmit": true`, `"target": "es2015"`
- Rename the `index.js` to `index.tsx` and update `public/index.html` accordingly
- Add this point, you should be able to start the project again, and the project should be running.
- It is time to fix all the compile errors, stop parcel and create the following script in `package.json`: `"ts": "tsc --watch --pretty --noemit"`. You can now run `yarn ts` and from now on can see all compile errors pop up in the the terminal. At this moment their should be none, as all files are still `.js`.
- This projects uses several libraries for which external typings exist. Run `yarn add -D @types/react @types/react-dom @types/jest`.
- Let's make sure the entire project becomes valid TypeScript code. Rename the source files from `.js` to

Solution

spoiler

Exercise 1

```
interface Circle {
  readonly id: string
  readonly x: number
  readonly y: number
}

type AppState = {
  readonly [key: string]: Circle
}
```

```
type MoveAction = {
  type: "move"
  id: string
  deltaX: number
  deltaY: number
}

type AddAction = {
  type: "add"
  id: string
  x: number
  y: number
}

type AppActions = MoveAction | AddAction
```

Exercise 2

package.json

```
@types/jest
@types/react
@types/react-dom
ts-jest
typescript
```

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "allowJs": false,
    "jsx": "react",
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "strictBindCallApply": true,
    "strictPropertyInitialization": true,
    "noImplicitThis": true,
    "alwaysStrict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
```

Advanced TypeScript

Meta Programming: Programming with types

All the fun stuff, that you won't need on a daily basis.

Toolbelt

- Generics
- `any`, `unknown`, `never`
- Literal types
- `keyof`
- Mapped types
- Conditional types

`any`, `unknown` and `never` types

any

```
let x: any = "four"
let y: number = 3

x = y // ok?
y = x // ok?
```

unknown

```
let x: unknown = "four"
let y: number = 3

x = y // ok?
y = x // ok?
```

`any` is for the lazy. `unknown` is for the skeptic.

never

```
function print(value: string | number) {
  if (typeof value === "string") {
    console.log("String: " + value)
  } else if (typeof value === "number") {
    console.log("Number: " + value)
  } else {
    console.log("Something else: " + value)
  }
}
```

? What is the inferred type for `value` in the last `else`?

The types of object keys

```
type Todo = {
  title: string
  done: boolean
}

const todo = {
  title: "Learn keyof",
  done: false
}

const value1 = todo.title
const value2 = todo["title"]

const property = "title"
const value3 = todo[property]
```

? What are the inferred types for `value1`, `value2` and `value3`

The types of object keys

```
type Todo = {
  title: string
  done: boolean
}

const todo = {
  title: "Learn keyof",
  done: false
}

type TodoPropertyName = "title" | "done"

let property: TodoPropertyName = "done"
const value3 = todo[property]
```

Keyof

```
type Todo = {
  title: string
  done: boolean
}

const todo = {
  title: "Learn keyof",
  done: false
}

type TodoPropertyName = keyof Todo

let property: TodoPropertyName = "done"
const value3 = todo[property]
```

Tip: typeof

💡 `typeof` can infer the type of a variable, function, etc...

```
const todo = {
  title: "Learn keyof",
  done: false
}

type TodoPropertyName = keyof typeof todo

let property: TodoPropertyName = "done"
const value3 = todo[property]
```

Keyof

```
function pluck<ITEM>(items: ITEM[], key: keyof ITEM): any[] {
  return items.map(item => item[key])
}

const todos = [{ title: "Learn keyof", done: true }, { title: "Learn mapped types", done: fal

pluck(todos, "title") // ["Learn keyof", "Learn mapped types"] // -> string[]
pluck(todos, "done") // [true, false] // -> boolean[]
pluck(todos, "author") // error!
```

Mapped types

```
function pluck<ITEM>(items: ITEM[], key: keyof ITEM): any[]
                                                       ^ yikes!
```

Mapped types

Extract generic argument

```
function pluck<ITEM, PROP>(items: ITEM[], key: PROP): (ITEM[PROP])[]
```

Mapped typed: iterating properties

```
type CloneType<BASE> = {
  [PROP in keyof BASE]: BASE[PROP]
}
```

Mapped typed: iterating

```
type Readonly<T> = {
  readonly [P in keyof T]: T[P]
}

type Record<K extends keyof any, T> = {
  [P in K]: T
}
```

Conditional types

Conditional types

Remember:

```
function first<ITEM_TYPE>(thing: ITEM_TYPE[]): ITEM_TYPE
function first<ITEM_TYPE>(thing: Map<any, ITEM_TYPE>): ITEM_TYPE
function first(thing: unknown): any {
  if (Array.isArray(thing))
    return thing[0]
  else if (thing instanceof Map && thing.size > 0)
    return thing.values().next().value
  return undefined
}

const a = first(["Noa", "Veria"])
const b = first(new Map([["Noa", 7], ["Veria", 5]]))
```

Let's eliminate the overloading!

Conditional types

```
const result = first([1, 2, 3]) // result should be inferred to be `number`!
```

Conditional types

```
type ExtractItemType<T> = T extends Map<any, infer U>
  ? U
  : T extends Array<infer U>
    ? U
    : never
```

Conditional types

```
type ExtractItemType<T> = T extends Map<any, infer U>
  ? U
  : T extends Array<infer U>
    ? U
    : never

function first<COLLECTION extends any[] | Map<any, any>(thing: COLLECTION): ExtractItemType<C
  //
}

const a = first(["Noa", "Veria"])  // string
const b = first(new Map([["Noa", 7], ["Veria", 5]])) // [string, number]
const c = first("nonsense") // compile error
```

Built-in utility types

```
/**
 * Make all properties in T optional
 */
type Partial<T> = {
    [P in keyof T]?: T[P]
}
```

```
type Partial<{
    title: string
    done: boolean
}> ⇒ /* becomes */ {
    title?: string
    done?: boolean
}
```

Built-in utility types

```ts
/**
 * Make all properties in T required
 */
type Required<T> = {
    [P in keyof T]-?: T[P];
};
//                ^ drops optional modifier


/**
 * Make all properties in T readonly
 */
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};
//      ^ adds readonly modifier


/**
 * From T, pick a set of properties whose keys are in the union K
 */
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
};
```

Exercises part 4

Use https://stackblitz.com/edit/typescript-8xntyp?file=index.ts and follow the instructions on the next slides.

Exercise `part4/omit`

Implement `Omit`

This is a real life example of a trick needed to use a library correctly that wasn't designed to be type friendly

In this exercise you will need to use several built-in types.

- Definitions
- Documentation

The goal of this exercise is: We want to reuse an existing type (`WebDriver.Options`), however, we want to override one specific field (`capabilities`) with a better type (based on `WebDriverHooks`). To achieve that, you need to implement `Omit`, so that we can take the definition of `WebDriver.Options`, but exclude the `capabilities` field!

In hindsight, would you have gone through the effort of figuring that out, or would you just have used `any`?

Bonus exercise: `part4/empty-object`

Implement the type for empty objects.

Go next for a hint.

Bonus exercise part4/json-verifier-part1

In this exercise we will be creating a static- and runtime-type checker for JSON data. The typechecker will give us some tools to describe the shape of a json structure at runtime, such that we can do things like:

```
const todoType = json.object({
  id: json.number,
  title: json.string,
  author: json.or(
    json.undefined,
    json.object({
      name: json.string
    })
  )
})

console.log(todoType({ nonsense: true })) // false, does not conform the given shape

console.log(
  todoType({
    id: 3,
    title: "test",
    author: {
      name: "You"
    }
  })
) // true, object matches the given shape
```

Solution

spoiler

Exercise 1

```ts
type Omit<T, K extends keyof T> = Pick<T, Exclude<keyof T, K>>

// Omit<{ title: string, done: boolean, author: string}, "title">

// keyof T
//     → "title" | "done" | "author"
// Exclude<"title" | "done" | "author", "title">
//     → "done" | "author"
// Pick<{ title: string, done: boolean, author: string }, "done" | "author">
//     → { done: boolean, author: string }
```

Exercise 2

```ts
type NotEmpty<T> = keyof T extends never ? never : T
type NotArray<T> = T extends any[] ? never : T

/* Make sure empty objects are not allowed
  Hint: use 'never', 'keyof' and '&' to 'include' impossible types
*/
function printKeys<T>(object: T & NotArray<T> & NotEmpty<T>) {
  console.log(Object.keys(object).join(", "))
}
```

Exercise 3

```
type JsonTypeChecker<T> = (val: T) ⇒ val is T

type MapLikeObject<T> = { [key: string]: T }

type JsonObjectType = {
  [key: string]: JsonTypeChecker<any>
}
type ExtractObjectShape<T extends JsonObjectType> = {
  [K in keyof T]: T[K] extends JsonTypeChecker<infer X> ? X : never
}

const json = {
  string(val: string): val is string {
    return typeof val === "string"
  },

  boolean(val: boolean): val is boolean {
    return typeof val === "boolean"
  },

  number(val: number): val is number {
    return typeof val === "number"
  }

  null(val: null): val is null {
    return val === null
```

Uncovered language features

Enums

```
enum Direction {
  Up,
  Down,
  Left,
  Right
}

const direction: Direction = Direction.Left
console.log(direction) // 2
```

Enums with static initializers

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT"
}

const direction: Direction = Direction.Down
console.log(direction) // "DOWN"
```

Enums versus types

```
type Direction = "UP" | "DOWN" | "LEFT" | "RIGHT"

const direction: Direction = "DOWN"
console.log(direction) // "DOWN"
```

Enums versus classes

```
class Direction {
  static Up = new Direction("UP")
  static Down = new Direction("DOWN")
  static Left = new Direction("LEFT")
  static Right = new Direction("RIGHT")

  private constructor(public value: string) {}

  toString() { return value }
```

Constructor types

```
class Document extends BaseAsset {}

function createNewDocument(type: new () ⇒ BaseAsset): BaseAsset {
  const res = new type()
  res.author = "You"
  return res
}

const myDoc = createNewDocument(Presentation) // → BaseAsset
```

`new (...args) => type` is the type of a constructor function, indication the function needs to be "newed".

Constructor types + generics

```
abstract class BaseAsset {
  author?: string
}
class Spreadsheet extends BaseAsset {}
class Presentation extends BaseAsset {}
class Document extends BaseAsset {}

function createNewDocument<T extends BaseAsset>(type: new () ⇒ T): T {
  const res = new type()
  res.author = "You"
  return res
}

const myDoc = createNewDocument(Presentation) // Presentation
```

`this` as return type

```
abstract class BaseAsset {
  clear(): BaseAsset {
    // do something useful
    return this
  }
}
class Spreadsheet extends BaseAsset {
  add(slideTitle: string): Spreadsheet {
    // do something useful
    return this
  }
}

new Spreadsheet().add("hello").add("world")
new Spreadsheet()
  .clear()
  .add("hello")
  .add("world")
```

? What compile error will this generate?

`this` as return type

```
abstract class BaseAsset {
  clear(): this {
    // do something useful
    return this
  }
}
class Spreadsheet extends BaseAsset {
  add(slideTitle: string): this {
    // do something useful
    return this
  }
}

new Spreadsheet().add("hello").add("world")
new Spreadsheet()
  .clear()
  .add("hello")
  .add("world")
```

`this` as parameter type

```
const reaction = new Reaction(
  name,
  function(this: Reaction) {
    this.track(reactionRunner)
  },
  opts.onError
)
```

typeof operator

```
interface Todo {
  title: string
  done: boolean
}

const myFirstTodo = {
  title: "test",
  done: false
}

function printTodo(todo: Todo) {
  console.log(`${todo.done ? "[x]" : "[ ]"} ${todo.title}`)
}

printTodo(myFirstTodo)
```

typeof operator

```
const myFirstTodo = {
  title: "test",
  done: false
}

function printTodo(todo: typeof myFirstTodo) {
  console.log(`${todo.done ? "[x]" : "[ ]"} ${todo.title}`)
}

printTodo({
  title: "second!",
  done: true
})
```

Functions are objects too

```
function overloaded(message: string): void
function overloaded(amount: number, error: Error): Error
function overloaded( ...args: any): any {
  throw "not implemented"
}

type FnType = typeof overloaded
```

`readonly` and `as const`

- Read only data structures: `readonly string[]` instead of `ReadonlyArray<string>` (arrays, tuples, objects)
- Read only literals: `{ hello: "world" } as const`
- Better literal types: `let propertyName = 'title' as 'title'` becomes: `let propertyName = 'title' as const`

Tip: re-exposing unpublished types

```
import { GraphQLClient } from "graphql-request"

//             ^ No Options exposed for the GraphQLClient constructor

export function createHttpClient(url: string, options?: WHAT) {
  return new GraphQLClient(url, options)
}
```

Tip: spec.ts

https://github.com/aleclarson/spec.ts

Make sure that the types exposed / inferred are what you expect them to be (and not accidentally `any`):

```
import { fancyMap } from "../src/fancyMap"
import { assert, _ } from "spec.ts"

assert(fancyMap([1, 2, 3], nr ⇒ "" + nr), _ as string[])
```

Bonus: even fancier meta programming

Revisiting assign

```
function assign<A, B, C, D>(obj1: A, obj2: B, obj3: C, obj4: D): A & B & C & D
function assign<A, B, C>(obj1: A, obj2: B, obj3: C): A & B & C
function assign<A, B>(obj1: A, obj2: B): A & B
function assign( ...objects: any[]): any {
  throw "Not implemented" // ignore this line
}
```

Revisiting assign

```ts
type Head<T extends any[]> = T extends [any, ...any[]] ? T[0] : never

// type Head<[string, number, boolean]> → string
// type Head<[number]> → number
// type Head<[]> → never
```

Revisiting assign

```
type Tail<T extends any[]> = T extends [any, ...infer REST] ? REST : []
```

Revisiting assign

```ts
type HasTail<T extends any[]> = T extends ([] | [any]) ? false : true

// type HasTail<[string, number, boolean]> → true
// type HasTail<[number]> → false
// type HasTail<[]> → false
```

Revisiting assign

```
// Turns tuples like [A, B, C] into A & B & C....
type AssignType<T extends any[]> = HasTail<T> extends true ? Head<T> & AssignType<Tail<T>> :

// type AssignType<[string, number, boolean]>
//     → string & AssignType<[number, boolean]>
//     → string & number & AssignType<[boolean]>
//     → string & number & boolean
```

Revisiting assign

```ts
// Turns tuples like [A, B, C] into A & B & C....
type AssignType<T extends any[]> = {
  0: Head<T> & AssignType<Tail<T>>
  1: Head<T>
}[HasTail<T> extends true ? 0 : 1]

// type AssignType<[string, number, boolean]>
//      → string & AssignType<[number, boolean]>
//      → string & number & AssignType<[boolean]>
//      → string & number & boolean
```

Revisiting assign

```
type Head<T extends any[]> = T extends [any, ...any[]] ? T[0] : never

type Tail<T extends any[]> = ((...t: T) => any) extends ((_: any, ...tail: infer TT) => any)

type HasTail<T extends any[]> = T extends ([] | [any]) ? false : true

// Turns tuples like [A, B, C] into A & B & C....
type AssignType<T extends any[]> = {
  0: Head<T> & AssignType<Tail<T>>
  1: Head<T>
}[HasTail<T> extends true ? 0 : 1]

// Correctly typed for any length of T!
function assign<T extends any[]>(...objects: T): AssignType<T> {
  throw "Not implemented" // ignore this line
}
```

Inspiration: https://medium.freecodecamp.org/typescript-curry-ramda-types-f747e99744ab

Q & A

Some great resources:

- http://2ality.com/2018/04/type-notation-typescript.html (Great summary of most important language constructs)
- https://basarat.gitbooks.io/typescript/ (Lot of best practices)
- https://mariusschulz.com/blog (pretty in-depth)
- https://github.com/sw-yx/react-typescript-cheatsheet (React + TS tips)

Feedback needed! info@michel.codes