Typescript 2024 Notes (incl. React, Redux, RTK) – Colt Steele & Stephen Grider

Table of Contents

[Typescript 2024 Notes (incl. React, Redux, RTK) – Colt Steele & Stephen Grider 1](#_Toc169358431)

[Introduction 8](#_Toc169358432)

[Installation and setup 8](#_Toc169358433)

[Type Annotation Basics 9](#_Toc169358434)

[Types in typescript 9](#_Toc169358435)

[Annotating a variable 9](#_Toc169358436)

[Type inference 9](#_Toc169358437)

[the any type 9](#_Toc169358438)

[delayed initialization and implicit any 10](#_Toc169358439)

[Functions 10](#_Toc169358440)

[Function parameter annotations 10](#_Toc169358441)

[Working with default parameters 10](#_Toc169358442)

[Return type annotations 10](#_Toc169358443)

[Anonymous functions 11](#_Toc169358444)

[Void 11](#_Toc169358445)

[The Never type 11](#_Toc169358446)

[Object Types 11](#_Toc169358447)

[Working with object types 11](#_Toc169358448)

[Excess properties 12](#_Toc169358449)

[creating type aliases with type keyword 12](#_Toc169358450)

[Nested objects 13](#_Toc169358451)

[optional object properties with ? 13](#_Toc169358452)

[The readonly modifier 13](#_Toc169358453)

[intersection types 14](#_Toc169358454)

[3 ways of accessing nested object properties 14](#_Toc169358455)

[Array Types 15](#_Toc169358456)

[practice exercise: 16](#_Toc169358457)

[Union Types 16](#_Toc169358458)

[Introducing union types 16](#_Toc169358459)

[union types and functions – type narrowing 17](#_Toc169358460)

[union types and arrays 17](#_Toc169358461)

[literal types 17](#_Toc169358462)

[Practice Exercise 18](#_Toc169358463)

[Tuples and Enums 19](#_Toc169358464)

[Tuples Introduction 19](#_Toc169358465)

[problems with tuples 19](#_Toc169358466)

[Enums introduction 20](#_Toc169358467)

[Enums behind the scenes – what they compile to 21](#_Toc169358468)

[Interfaces 21](#_Toc169358469)

[Introduction to interfaces and interface keyword 21](#_Toc169358470)

[Readonly and optional (? keyword) Interface properties 21](#_Toc169358471)

[interface methods 21](#_Toc169358472)

[interface method parameters 22](#_Toc169358473)

[Reopening interfaces 22](#_Toc169358474)

[Extending interfaces with the extends keyword 23](#_Toc169358475)

[Interface multiple inheritance 23](#_Toc169358476)

[Interfaces vs Type Aliases 23](#_Toc169358477)

[The Typescript Compiler – configuring typescript 24](#_Toc169358478)

[Compiling to javascript 24](#_Toc169358479)

[watch mode 24](#_Toc169358480)

[Working with multiple files 24](#_Toc169358481)

[the files compiler option 24](#_Toc169358482)

[Include and exclude options 24](#_Toc169358483)

[outDir option 25](#_Toc169358484)

[Target option 25](#_Toc169358485)

[Strict option 26](#_Toc169358486)

[other options 26](#_Toc169358487)

[Mini project – the DOM, type assertions and more 26](#_Toc169358488)

[Project setup 26](#_Toc169358489)

[working with the DOM 27](#_Toc169358490)

[the lib compiler option 27](#_Toc169358491)

[Typescript’s Non-null assertion operator 27](#_Toc169358492)

[Type assertions 28](#_Toc169358493)

[Handling events 28](#_Toc169358494)

[building the todo list 28](#_Toc169358495)

[connecting with local storage 29](#_Toc169358496)

[Completed project - index.js 30](#_Toc169358497)

[(JS Classes) 31](#_Toc169358498)

[The class keyword 31](#_Toc169358499)

[Contructors 31](#_Toc169358500)

[Class fields 32](#_Toc169358501)

[Private fields (properties) 32](#_Toc169358502)

[Getters 33](#_Toc169358503)

[Setters 33](#_Toc169358504)

[Static properties & methods 34](#_Toc169358505)

[Extending classes/inheritance 35](#_Toc169358506)

[Super 35](#_Toc169358507)

[Typescript classes 35](#_Toc169358508)

[Annotating classes in typescript 35](#_Toc169358509)

[class fields 35](#_Toc169358510)

[Readonly modifier/class properties 36](#_Toc169358511)

[The public modifier 36](#_Toc169358512)

[The private modifier 36](#_Toc169358513)

[Parameter properties shorthand 37](#_Toc169358514)

[Getters and setters 37](#_Toc169358515)

[The protected modifier 37](#_Toc169358516)

[Classes and interfaces 38](#_Toc169358517)

[Creating abstract classes 38](#_Toc169358518)

[Generics 39](#_Toc169358519)

[Introducing generics 39](#_Toc169358520)

[Another example of a built-in generic function 40](#_Toc169358521)

[Writing our first generic function 40](#_Toc169358522)

[Writing another generic function 40](#_Toc169358523)

[Inferred generic type patterns 41](#_Toc169358524)

[Generics, arrow functions & tsx files 41](#_Toc169358525)

[Generics with multiple types 41](#_Toc169358526)

[Adding type constraints 41](#_Toc169358527)

[Default type parameters 42](#_Toc169358528)

[Writing generic classes 42](#_Toc169358529)

[Type Narrowing 42](#_Toc169358530)

[typeof guards 42](#_Toc169358531)

[truthiness guards 43](#_Toc169358532)

[equality narrowing 43](#_Toc169358533)

[narrowing with the In Operator 44](#_Toc169358534)

[instanceof narrowing 45](#_Toc169358535)

[working with type predicates 45](#_Toc169358536)

[discriminated unions 46](#_Toc169358537)

[exhaustiveness checks with never 47](#_Toc169358538)

[Working with Type Declarations 47](#_Toc169358539)

[Introducing type declarations 47](#_Toc169358540)

[Using 3rd party libraries – axios 48](#_Toc169358541)

[Working with axios types 48](#_Toc169358542)

[Installing types separately 49](#_Toc169358543)

[Modules 49](#_Toc169358544)

[A note on namespaces 50](#_Toc169358545)

[Working without modules 50](#_Toc169358546)

[Using typescript modules 50](#_Toc169358547)

[Changing compilation module system 51](#_Toc169358548)

[Import/export syntax in depth 51](#_Toc169358549)

[Importing types 51](#_Toc169358550)

[Webpack & Typescript 52](#_Toc169358551)

[What’s the point of webpack 52](#_Toc169358552)

[Setting up a project 52](#_Toc169358553)

[Installing webpack dependencies 52](#_Toc169358554)

[Basic webpack config 52](#_Toc169358555)

[Adding source maps 53](#_Toc169358556)

[Fixing the missing devServer config 53](#_Toc169358557)

[Webpack dev server 54](#_Toc169358558)

[Production configuration 54](#_Toc169358559)

[colt’s webpack course 55](#_Toc169358560)

[React & Typescript 55](#_Toc169358561)

[Introduction 55](#_Toc169358562)

[Vite + React + Typescript 55](#_Toc169358563)

[Create react app with typescript 56](#_Toc169358564)

[Our first component 56](#_Toc169358565)

[A note on React functional components 56](#_Toc169358566)

[Props with typescript 56](#_Toc169358567)

[Creating our ShoppingList Component 57](#_Toc169358568)

[useState with TypeScript 58](#_Toc169358569)

[creating our ShoppingItem Form 58](#_Toc169358570)

[useRef with Typescript 59](#_Toc169358571)

[updating the state 60](#_Toc169358572)

[finishing touches 60](#_Toc169358573)

[React + Typescript wrapup 61](#_Toc169358574)

[Typescript + React 61](#_Toc169358575)

[Introduction 61](#_Toc169358576)

[Child props 62](#_Toc169358577)

[3 different approaches to applying the interface 62](#_Toc169358578)

[Property 'children' does not exist 62](#_Toc169358579)

[How to pass a function from parent to child 63](#_Toc169358580)

[Annotations with children 63](#_Toc169358581)

[State with typescript 64](#_Toc169358582)

[Type inference with state 64](#_Toc169358583)

[Example 1 64](#_Toc169358584)

[Example 2 65](#_Toc169358585)

[Typing standalone event handlers 66](#_Toc169358586)

[Other type of events -drag, form submission events 66](#_Toc169358587)

[Applying types to Refs 67](#_Toc169358588)

[Typescript + Redux 67](#_Toc169358589)

[Installation 67](#_Toc169358590)

[searching npm 68](#_Toc169358591)

[file structure 68](#_Toc169358592)

[Applying typescript to the reducer – annotating the return type 68](#_Toc169358593)

[typing an action 69](#_Toc169358594)

[separate interfaces for actions 69](#_Toc169358595)

[Better way to type actions 70](#_Toc169358596)

[adding an action type enum 71](#_Toc169358597)

[Refactoring into different files & directories 72](#_Toc169358598)

[annotating try catch block/ adding action creators & request logic 74](#_Toc169358599)

[setting up exports 75](#_Toc169358600)

[wiring up to React 75](#_Toc169358601)

[calling an action creator 76](#_Toc169358602)

[setting up a useActions hook to simplify the dispatch call 76](#_Toc169358603)

[Using the useActions hook 77](#_Toc169358604)

[useSelector, React-redux and typing 77](#_Toc169358605)

[setting up and using useTypedSelectorHook 78](#_Toc169358606)

[Big issues with Redux/React-Redux + Typescript 79](#_Toc169358607)

[Typescript + Redux Toolkit 79](#_Toc169358608)

[Installation 79](#_Toc169358609)

[Part 1 – Set up Redux Toolkit 79](#_Toc169358610)

[1. Create a redux store 79](#_Toc169358611)

[2. Provide the redux store to React 80](#_Toc169358612)

[3. Create a redux state slice 80](#_Toc169358613)

[4. Add slice reducers to the store 81](#_Toc169358614)

[5. use redux state and actions in React components 81](#_Toc169358615)

[Part 2 – Add typescript 82](#_Toc169358616)

[6. Define Root State and Dispatch Types 82](#_Toc169358617)

[7. Define typed hooks 82](#_Toc169358618)

[8. Application usage – define slice state and action types 83](#_Toc169358619)

[9. Use typed hooks in components 84](#_Toc169358620)

[Further Details 84](#_Toc169358621)

[ChatGPT Explanation: Typescript + React + Redux Toolkit 84](#_Toc169358622)

[Step 1: Setting Up a TypeScript React Application 85](#_Toc169358623)

[Step 2: Install Redux Toolkit and Dependencies 85](#_Toc169358624)

[Step 3: Define Redux Slices with TypeScript 85](#_Toc169358625)

[Step 4: Create Redux Store with TypeScript 86](#_Toc169358626)

[Step 5: Set Up Redux Provider in Your App 86](#_Toc169358627)

[Step 6: Use Redux State and Actions in TypeScript Components 87](#_Toc169358628)

[Step 7: Run Your Application 87](#_Toc169358629)

[Typescript + React + Redux Toolkit + Redux Toolkit Query 88](#_Toc169358630)

[ChatGPT explanation 88](#_Toc169358631)

[Step 1: Setting Up a TypeScript React Application 88](#_Toc169358632)

[Step 2: Install Dependencies 88](#_Toc169358633)

[Step 3: Define Redux Slices with TypeScript 89](#_Toc169358634)

[Step 4: Create Redux Store with TypeScript 89](#_Toc169358635)

[Step 5: Define RTK Query API Service with TypeScript 90](#_Toc169358636)

[Step 6: Set Up Redux Provider in Your App 90](#_Toc169358637)

[Step 7: Use Redux State, Actions, and RTK Query in TypeScript Components 91](#_Toc169358638)

# Introduction

* JS is a pre-requisite
* Need to know Variables, arrays, objects, functions, conditionals, loops
* Covered in the course: classes, the DOM, React, webpack
* What is typescript?
  + Needed because javascript is quirky, e.g. null \* 123 returns 0
  + JS wasn’t created to do what it does today
  + Fixes and enhancements haven’t fixed the issues
  + Typescript is built ontop of js
  + It adds types to improve js experience
  + Ts is a superset of js
* What are types?
  + Static checking – ts detects errors in our code without running it
  + Ts is a static error checker - It does error checking on the basis of the kind of data (types) in our program
  + Helps us find errors before the code runs
  + Analyses our code as we type
  + Only exists in development
* Workflow
  + We type ts code
  + We fix errors it identifies
  + Code is then compiled to regular js

# Installation and setup

* Pre-requisite: node
* Install by project:
  + npm install typescript --save-dev
* install globally:
  + npm install -g typescript
* Typescript playground: <https://www.typescriptlang.org/play>
* Use Playground > examples to understand ts code
* Use playground > ts config to configure ts
* Vscode extensions (not used in this course but are available)
  + Tslint
  + Tslint snippets
* Files must end in .ts (.tsx in react)
* 2 steps of using ts:
  + we type the code and typescript will give errors that we can correct
  + we compile into js:
    - in the terminal we run:
    - tsc basics.ts
    - this will create basics.js
    - we then execute basics.js
    - node basics.js

# Type Annotation Basics

## Types in typescript

* primitive types
  + number
  + string
  + boolean
  + null
  + undefined
  + void
  + any
  + never
  + unknown
* Object types
  + object
  + array
  + function
  + tuple
  + enum
  + others!

## Annotating a variable

* syntax:
  + let myVar**: type** = value
  + e.g. let myString**: string** = ‘hello’
  + if you try to set that variable to a number, ts will give an error
* compiling ts:
  + strips out the ts syntax
  + gives errors but doesn’t stop compiling the js code and producing a js file (with errors)

## Type inference

* type annotation is not needed most of the time because of type inference
* type inference refers to the typescript compilers ability to infer types from certain values in your code, e.g. let x = 7; typescript thinks x’s type is a number
* typescript can remember a value’s type even if you didn’t provide a type annotation and it will enforce that type moving forward

## the any type

* any is an escape hatch: it turns off type checking for this variable so that it can be anything
* it doesn’t check the type or if it’s a valid function or object property
* use sparingly

## delayed initialization and implicit any

* if a variable is declared but not initialized, e.g. let foundMovie; typescript infers its type is any
* that could potentially introduce errors
* to prevent errors, we can annotate a delayed initialized variable: let foundMovie: string;

# Functions

## Function parameter annotations

* we can specify the type of function parameters in a function definition
* then ts will enforce types for the values being passed to your function
* same as typing variables
* if no type is specified the default type for function parameters is any

const encourageStudent = (name: string) => {

return `Hey, ${name}, you’re doing GREAT!`;

}

encourageStudent(‘Jimbob’) // ‘Hey JimBob, you’re doing GREAT!`

encourageStudent(61) //errors

* what if you want multiple types (e.g. a Boolean or a string)?
* What if you want to type a object property that you’ve destructured e.g.

function MyComponent ({isEnabled: boolean, isLoggedIn: boolean}) {

//jsx goes here

}

* With multiple parameters, if all arguments (e.g. function has 3 parameters but you only call it with one argument) are not provided or the incorrect types are given then ts will error

## Working with default parameters

* Default parameter needs to come after the ts annotation

function greet (person: string = ‘stranger’) {

return `Hi there ${person}!`;

}

## Return type annotations

* We can specify the type returned by a function. Even though ts can infer this (it calculates what the return type should be if you don’t put the return type in), colt prefers explicit annotations
* Add the type annotation after the function parameter list

const addNums = (x: number, y: number): number => {

return x + y;

}

* If the return keyword is omitted, the return type will be void
* If the function can return a string or number, typescript will infer :string | number (union type)
* Works with all types of functions, e.g. arrow functions:

const add = (x: number, y: number): number => {

return x + y;

};

* Its helpful to include return types as then you can easily see what a function returns

## Anonymous functions

* E.g. for each, map

const colors = [‘red’, ‘orange’, ‘yellow’]

colors.map(color => { color.toUpperCase()})

* Typescript infers the type of color (string)
* It will give an error if you run color.toFixed()

## Void

* Used with functions not really with variables
* Used to say this function shouldn’t return anything
* You can explicitly annotate a function return type as void so that typescript knows the function shouldn’t return anything (ts will error if you try to return something from that function):

function printTwice(msg: string): void {

console.log(msg)

console.log(msg)

return ‘hello’ //gives error

}

## The Never type

* The never type represent values that never occur
* Used to show a function that never returns or a function that loops forever/never stops
* E.g. when you throw an error or run an infinite loop

function makeError(msg: string): never {

throw new Error(msg);

}

function gameLoop():never {

while (true) {

console.log(‘GAME LOOP RUNNING!’);

}

}

* Don’t confuse never with void. Void returns undefined or null which is technically still a value, whilst with never the function doesn’t finish executing

# Object Types

## Working with object types

* Objects can be typed by declaring what the object should look like in the annotation
* Accessing a property that isn’t defined or performing operations without the correct types will throw errors

function printName(person: { first: string, last: string}):void {

console.log(`${person.first} ${person.last}`)

}

* Can do this for a function parameter, a return type or a variable

let coordinate: {x: number, y: number } = {x : 34, y: 2}

function randomCoordinate(): {x: number, y: number} {

return {x: Math.random(), y: Math.random() }

}

## Excess properties

* When ts was written they made the decision to give an error if excess object properties were specified than those that were annotated. However, it does not error if you pass in a variable with excess properties

function printName(person: { first: string, last: string}):void {

console.log(`${person.first} ${person.last}`)

}

printName({first: ‘Mick’, last: ‘Jagger’, age: 473}) //gives error about age

const singer = {first: ‘Mick’, last: ‘Jagger’, age: 473, isAlive: true}

printName(singer) // however this works! Excess properties are ignored

## creating type aliases with type keyword

* Instead of writing our objects in an annotation we can declare them separately in a type alias with a name
* A type alias is the desired shape of the object
* Use type keyword, the convention is to use a capital letter for the name, and separate properties with semi-colon
* Note: type aliases are stored in the global namespace so they have to be named uniquely; if you repeat a type with the same name, ts will error (interfaces however can be repeated, see ‘reopening interfaces’)
* This makes code more readable and we can reuse the types elsewhere in the code
* Helpful for object types

type Point = {

x: number;

y: number;

}

let coordinate: Point = {x: 34, y: 2}

function randomCoordinate(): Point {

return { x: Math.random(), y: Math.random()}

}

function doublePoint(point: Point) : Point {

return {x: point.x \* 2, y: point.y \* 2};

}

## Nested objects

* We can annotate nested objects with types

type Song = {

title: string;

artist: string;

numStreams: number;

credits: { producer: string; writer: string };

};

function calculatePayout(song: Song): number {

return song.numStreams \* 0.0033

}

function printSong( song: Song): void {

console.log(`${song.title} - ${song.artist}`)

}

const mySong: Song = {

title: ‘Unchained Melody’,

artist: ‘Righteous Brothers’,

numStreams: 12873321,

credits: { producer: ‘Phil Spector’, writer: ‘Alex North’},

};

calculatePayout(mySong);

printSong(mySong);

## optional object properties with ?

type Point = {

x: number;

y: number;

z?: number;

}

* ? makes z optional, if you use the Point type and z is missing it wont complain

## The readonly modifier

* readonly keyword is a modifier used in ts, we can mark object properties as read-only

type User = {

readonly id: number;

username: string;

}

const user: User = {

id : 12345,

username: ‘Catgurl’

}

console.log(user.id) // works

user.id = 67890 // will error

## intersection types

* a way to combine type aliases

type Circle = {

radius: number;

}

type Colorful = {

color: string;

}

type ColourfulCircle = Circle & Colorful

const smileyFace: ColourfulCircle = {

radius: 4,

color: ‘yellow’

}

* how to add additional properties:

type Cat = {

numLives: number;

}

type Dog = {

breed: string;

}

type CatDog = Cat & Dog & { age: number; }

const misty: CatDog = {

numLives: 7,

breed: ‘husky’,

age: 9,

}

## 3 ways of accessing nested object properties

type Movie = {

readonly title: string;

originalTitle?: string;

director: string;

releaseYear: number;

boxOffice: {

budget: number;

grossUS: number;

grossWorldwide: number;

};

};

const dune: Movie = {

title: "Dune",

originalTitle: "Dune Part One",

director: "Denis Villeneuve",

releaseYear: 2021,

boxOffice: {

budget: 165000000,

grossUS: 108327830,

grossWorldwide: 400671789,

},

};

const cats: Movie = {

title: "Cats",

director: "Tom Hooper",

releaseYear: 2019,

boxOffice: {

budget: 95000000,

grossUS: 27166770,

grossWorldwide: 73833348,

},

};

function getProfit(movie: Movie): number {

return movie.boxOffice.grossWorldwide - movie.boxOffice.budget;

}

function getProfit2(movie: Movie): number {

const { grossWorldwide, budget } = movie.boxOffice;

return grossWorldwide - budget;

}

function getProfit3({ boxOffice: { grossWorldwide, budget } }: Movie): number {

return grossWorldwide - budget;

}

# Array Types

* arrays can be typed using a type annotation followed by empty array brackets, e.g:
* number[] for an array of numbers
* note: these arrays only allow data of one type inside them
* if you don’t specify the type of an array it will be type any

Examples:

const activeUsers: string[] = [];

activeUsers.push("brian");

//activeUsers.push(12); //errors

const ageList: number[] = [45, 56, 13];

ageList[0] = 99;

//ageList[1] = "hello"; //errors

const bools: boolean[] = [];

const bools2: Array<boolean> = [];

//alternative syntax

const names: Array<string> = ["bob", "tim", "jane", "bill"];

const ages1: Array<number> = [12, 34, 65, 23, 14];

//custom object types

type Point = {

x: number;

y: number;

};

//an array of type Point objects

const coords: Point[] = [];

coords.push({ x: 34, y: 2 });

//coords.push({ x: 3, y: "a" }); //errors

//multidimensional arrays

const board: string[][] = [

["X", "O", "X"],

["X", "O", "X"],

["X", "O", "X"],

];

const demo: number[][][] = [[[1]], [[2]], [[3]]];

## practice exercise:

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Create an empty array of numbers called "ages":

const ages: number[] = [];

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Create an array variable called gameBoard that starts as an empty array.

// It should be typed to hold a 2 dimensional array of strings

const gameBoard: string[][] = [];

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Create a Product type that contains a name and a price.

// An example product could be:

// {name: "coffee mug", price: 11.50}

type Product = {

name: string;

price: number;

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 4 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Write a function called getTotal that accepts an array of Product types

// It should return the sum of all the products' prices

function getTotal(myProducts: Product[]) {

let total = 0;

myProducts.forEach((product) => (total += product.price));

return total;

}

# Union Types

## Introducing union types

* allows to give multiple possible types
* let age: number | string |boolean = 21

type Point2 = {

x: number;

y: number;

};

type Loc = {

lat: number;

long: number;

};

let coordinates: Point2 | Loc = { x: 1, y: 34 };

coordinates = { lat: 321.213, long: 23.345 };

## union types and functions – type narrowing

* problem:

function calculateTax(price: number|string, tax:number) {

//price.replace('$', '')//errors

return tax \* price //errors

}

* type narrowing is where you use a typeof check to check if a variable is a string/number
* ts will not error if you check the type of a variable before you run your operation

function calculateTax2(price: number | string, tax: number) {

if (typeof price === "string") {

price = Number.parseFloat(price.replace("$", ""));

}

return tax \* price;

}

## union types and arrays

* to limit an array to multiple types, instead of using any use union

const nums: number[] = [1, 2, 3, 4];

const stuff: any[] = [1, 2, 3, 4, "a", true];

const stuff2: (number | string)[] = [1, 2, 3, "a"];

* a common mistake students make is the following: (which means either an array of strings or an array of numbers not both):

let stuff3: number[] | string[] = [1, 2, 3];

stuff3 = ["a", "b", "c"];

* an array of custom object types:

const coords: (Point2 | Loc)[] = [];

coords.push({ lat: 234, long: 123 });

coords.push({ x: 2, y: 4 });

## literal types

* literal types are not just types, but values themselves too
* combine it with unions and you have fine-tuned type options for ts to enforce

//0 or ‘hi’ is a literal type

let zero: 0 = 0

zero =2 /errors

let hi: ‘hi’ = ‘HI’ //errors

//literal types are useful when combined with union types

const giveAnswer = (answer: ‘yes’ | ‘no’ | ‘maybe’) => {

return `the answer is ${answer}.`;

}

giveAnswer(‘no’) //’the answer is no’

giveAnswer(‘oh boy im not sure’); //errors

//other examples

let mood: "Sad" | "Happy" = "Happy";

mood = "Sad";

type DayofWeek =

"Monday"

| "Tuesday"

| "Wednesday"

| "Thursday"

| "Friday"

| "Saturday"

| "Sunday";

let today: DayofWeek = "Monday";

today = "tues"; //errors

## Practice Exercise

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Create a variable called highScore that can be a number OR a boolean

let highScore: number | boolean;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// create an array called stuff

// it can be an array of numbers OR an array of strings

// it cannot be an array of numbers and strings (mixed together)

const stuff4: number[] | string[] = [1, 2, 4];

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Create a literal type called SkillLevel

// There are 4 allowed values: "Beginner", "Intermediate", "Advanced", and "Expert"

type SkillLevel = "Beginner" | "Intermediate" | "Advanced" | "Expert";

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 4 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Create a type called SkiSchoolStudent

// name must be a string

// age must be a number

// sport must be "ski" or "snowboard"

// level must be a value from the SkillLevel type (from above)

type SkiSchoolStudent = {

name: string;

age: number;

sport: "ski" | "snowboard";

level: SkillLevel;

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 5 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Define a type to represent an RGB color

// r should be a number

// g should be a number

// b should be a number

type RGB = {

r: number;

g: number;

b: number;

};

// Define a type to represent an HSL color

// h should be a number

// s should be a number

// l should be a number

type HSL = {

h: number;

s: number;

l: number;

};

// Create an array called colors that can hold a mixture of RGB and HSL color types

const colors: (RGB | HSL)[] = [];

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PART 6 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Write a function called greet that accepts a single string OR an array of strings

// It should print "Hello, <name>" for that single person OR greet each person in the array with the same format

function greet(person: string | string[]): void {

if (typeof person === "string") {

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

} else {

person.forEach((name) => console.log(`Hello, ${name}`));

}

}

# Tuples and Enums

## Tuples Introduction

* tuples are a special type exclusive to ts (they don’t exist in js)
* an array type of fixed length and ordered with specific types
* e.g. in the following example the first element has to be a number the second has to be a string

let myTuple: [number, string]

myTuple = [10, ‘typescript Is fun’] //this works

myTuple = [‘Typescript is fun’, 10]; //errors

* rgb color:

//an rgb color array (only allows 3 numbers):

const color: [number, number, number] = [100, 0, 100];

## problems with tuples

type HTTPResponse = [number, string]

const goodRes: HTTPResponse = [200, 'OK'];

//if you use push after the array has been created tuples will allow it

goodRes.push(123) //works even though it shouldn't

goodRes.push(true) //errors

goodRes.pop() //works even though it shouldn't

goodRes.pop() //works even though it shouldn't

goodRes.pop() //works even though it shouldn't

//a better solution

type HTTPResponse2 = {

code: number;

msg: string;

}

//an array of tuples

const responses: HTTPResponse[] = [

[404, "not found"],

[200, "ok"],

]; //works but if you add to it, ts will allow anything

//note to self: in my code editor ts errors if you deviate from the tuple structure

## Enums introduction

* dont exist in js but do exist in ts
* enums allow us to define a set of named constants; we can give these constants numeric or string values
* a set of different names we can refer back to
* e.g. responses (yes, no, maybe), arrowkeys(up, down, left, right) status(pending, shipped, delivered)
* making an enum automatically assigns numbers to each definition/label from 0 but you can assign numbers

enum OrderStatus {

PENDING =34,

SHIPPED=56,

DELIVERED =435,

RETURNED=3,

}

const myStatus = OrderStatus.DELIVERED;

function isDelivered(status: OrderStatus) {

return status === OrderStatus.DELIVERED;

}

isDelivered(OrderStatus.DELIVERED); //true

enum ArrowKeys {

UP = "up",

DOWN = "down",

LEFT = "left",

RIGHT = "right",

ERROR = 400,

}

## Enums behind the scenes – what they compile to

* enums compile to an object (check it out in TS playground)
* in stackover flow people say they avoid using enums for that reason and create their own objects
* however enums get autocomplete which makes them nice to use
* if you put const before the enum it will compile to just the number of the label

# Interfaces

## Introduction to interfaces and interface keyword

* it is very similar to type aliases, they serve the same purpose (with different syntax)
* allow us to describe the shape of objects (only)

interface Person {

name: string;

age: number;

}

const me: Person = { name: ‘Bill’, age: 12 }

## Readonly and optional (? keyword) Interface properties

* same syntax as was done before (readonly and ?)

interface Person {

readonly id: number;

first: string;

last: string;

nickname?: string;

}

const thomas: Person = {id: 1234, first: ‘Thomas’, last: ‘Hardy’, nickname: ‘Tom’}

## interface methods

* we can also define interface methods, here we have sayHi which accepts 0 arguments and returns a string

interface Person {

readonly id: number;

first: string;

last: string;

nickname?: string;

sayHi: () => string;

//sayHi(): string;// same as above line

}

const thomas: Person = {id: 1234, first: ‘Thomas’, last: ‘Hardy’, nickname: ‘Tom’, sayHi: ()=> { return ‘Hello!’}}

## interface method parameters

* we can specify parameters and their type for a method

interface Product {

name: string;

price: number;

applyDiscount(discount: number): number;

}

//applyDiscount method accepts a number, the name of the parameter does not matter - its just descriptive

// the interface doesnt say what the method does, just what it accepts and what it returns

const shoes: Product = {

name: "Blue Suede Shoes",

price: 100,

applyDiscount(amount: number) {

const newPrice = this.price \* (1 - amount);

this.price = newPrice;

return this.price;

},

};

shoes.applyDiscount(0.4);

## Reopening interfaces

* adding new properties to an interface
* to reopen the interface and add new properties to it, use the same interface name with the interface keyword
* used if the interface is in a 3rd party library or in another file

interface Dog {

name: string;

age: number;

}

//you can add on additional properties to the Dog interface by reopening it

interface Dog {

breed: string;

bark(): string;

}

const elton: Dog = {

name: "Elton",

age: 0.5,

breed: "Australian Shepherd",

bark: () => "woof",

};

## Extending interfaces with the extends keyword

* we can extend an interface e.g. inherit properties from another interface (similar to classes) with the extends keyword

interface ServiceDog extends Dog {

job: "drug sniffer" | "bomb" | "guide dog";

}

const chewy: ServiceDog = {

name: "Chewy",

age: 12,

breed: "German shepherd",

bark: () => "woof woof",

job: "guide dog",

};

## Interface multiple inheritance

* we can extend with multiple interfaces

interface Human {

name: string;

}

interface Employee {

readonly id: number;

email: string;

}

interface Engineer extends Human, Employee {

level: string;

languages: string[];

}

const pierre: Engineer = {

id: 1234,

name: "Pierre",

level: "junior",

languages: ["HTML", "CSS", "Javascript", "React", "Typescript"],

email: "pierre@gmail.com",

};

## Interfaces vs Type Aliases

* how do they compare and why would you use which one
* key differences:
  + interfaces can only describe object types. Alias types can describe type literals and other types
  + type aliases cannot reopened
  + type aliases cannot be extended (you have to use intersection types – the & keyword)

# The Typescript Compiler – configuring typescript

## Compiling to javascript

* run from command line:
  + cd into your project directory, then run:
  + tsc –init
  + this creates a tsconfig.json file
  + this file’s entries need double quotes as it’s a json file
* (Note: Vite + React + Typescript creates and configures its tsconfig.json file automatically, there is no need to edit it or run tsc)
* in the typescript playground > TS Config > you can see all the typescript configuration settings <https://www.typescriptlang.org/tsconfig/>
* to compile to javascript
  + tsc index.ts
  + creates an index.js file
  + you can run this in node:
  + node index.js

## watch mode

* watch mode will keep track of a file and recompile it if it changes
  + tsc -w <filename>
  + tsc –watch <filename>
* then you can keep editing the tsc file and the terminal will show errors or ok message

## Working with multiple files

* to compile more than one file run tsc on its own in the working directory
* can configure which files to compile in tsconfig.json
* run tsc -w to compile all files in watch mode

## the files compiler option

<https://www.typescriptlang.org/tsconfig#files>

* files is a root field, that means in tsconfig.json put ‘files’ option outside/after the compiler option
  + “files”: [“farmstand.ts”, “index.ts”]
  + Only files listed will be compiled when you run tsc, everything else will be ignored

## Include and exclude options

<https://www.typescriptlang.org/tsconfig#include>

* We can say which directories to include
* We can say which files to exclude
* The default is that tsc will compile all .ts files in all directories, unless ‘files’ is specified, in which case it will compile those files as well as whatever is listed in ‘include’
* These filenames are resolved relative to the directory containing the tsconfig.json file.

{

"include": ["src/\*\*/\*", "tests/\*\*/\*"]

}

* ‘exclude’ specifies an array of filenames or patterns that should be skipped when resolving include

{

exclude: [“test/\*\*.test.ts”, “node\_modules”]

}

* By default ts excludes node\_modules, but you need to add it in if you are using ‘exclude’
* It is not a mechanism that prevents a file from being included in the codebase - it simply changes what the include setting finds

## outDir option

<https://www.typescriptlang.org/tsconfig#outDir>

* The default output directory is alongside the ts file
* However in a project we would want to compile the js files to a /dist directory

{

"compilerOptions": {

“outDir”: “./dist”,

}

}

## Target option

<https://www.typescriptlang.org/tsconfig#target>

* It governs what version of js, ts will compile into
* Default: Default: ES3, Allowed: es3, es5, es6/es2015, es2016, es2017, es2018, es2019, es2020, es2021, es2022, esnext
* E.g. need to set it to es2015 if you want it to compile to js with arrow functions

{

"compilerOptions": {

"target": "es2016",

}

}

## Strict option

<https://www.typescriptlang.org/tsconfig#strict>

* Defaults to true
* Enable all strict type checking options
* Ensures stronger program correctness
* Options that it defaults to true include
  + noImplicitAny – errors if variable or parameter has an implicit any
  + strictNullChecks – errors if variable is set to null/undefined and it doesn’t have a null/undefined type
  + for full list of options see documentation/tsconfig file

## other options

* in "compilerOptions" other options include:
  + “lib” – will be covered later
  + “module“ – will be covered later
  + “allowJs” – allow js files to be part of program
  + “checkJs” – it will check js files and show errors if there are still ts problems
  + “sourceMap” –
  + “noEmit” – don’t output js files, but do the type checking
  + “noEmitOnError” – only compile js if ts is error free

# Mini project – the DOM, type assertions and more

## Project setup

* To setup initially:
  + tsc –init
  + mkdir src dist
  + touch src/index.ts
  + configure tsconfig.js
    - "outDir": "./dist",
    - "include": ["src/\*\*/\*"]
  + tsc -w
* to view in the browser
  + npm init -y
  + npm install lite-server
  + in package.json, scripts:
    - “start”: “lite-server”,
  + npm start

## working with the DOM

* by default, typescript knows about the DOM, document and its type
* right click on document in the ts file and choose go to type definition
* there you can see type definitions for that object
* e.g.: getElementById(elementId: string): HTMLElement | null;

## the lib compiler option

<https://www.typescriptlang.org/tsconfig#lib>

* TypeScript includes a default set of type definitions for built-in JS APIs (like Math), as well as type definitions for things found in browser environments (like document)
* TypeScript also includes APIs for newer JS features matching the target you specify; for example the definition for Map is available if target is ES6 or newer.
* how to turn typescript DOM types off:
  + default tsconfig.json:
    - // "lib": [], /\* Specify a set of bundled library declaration files that describe the target runtime environment. \*/
  + To turn off DOM types uncomment “lib” and set an empty array:
    - "lib": []
  + To enable DOM types manually put DOM in the array:
    - "lib": [ "DOM", "ES2018" ]
* Note type definition files are \*.d.ts
* You can view them on Github:
  + <https://github.com/microsoft/TypeScript/tree/main/src/lib>
* Using ES2021 target library – change the lib compiler option to ES2021
  + - "lib": [ "DOM", "ES2021" ]
    - Can still use “target”: “es5” if you want code to compile to es5

## Typescript’s Non-null assertion operator

* By default ts thinks that an element that is selected is of type HTMLElement | null
* This means that you cant run operations like addEventListener on btn
* To get around this you could put ? after btn

const btn = document.getElementById(“btn”);

btn?.addEventListener(“click”, function () {

console.log(“clicked”)})

* Or use non-null assertion operator ! to tell ts that element is not null

const btn = document.getElementById(“btn”)!

btn.addEventListener(“click”, function () {

console.log(“clicked”)})

* This is a last option as you want ts to validate variables (it could be null)

## Type assertions

* Sometimes you might have more specific information about a values type and you want ts to know this too
* You can assert a value’s type by using the as keyword followed by the specific type you want to assert

//typescript infers a type of HTMLElement

const myPic = document.querySelector(“profile-image”)

//but we know something more specific

const myPic = document.querySelector(“profile-image”) as HTMLImageElement

* Example:

const btn = document.getElementById("btn")! as HTMLButtonElement;

const input = document.getElementById("todoinput")! as HTMLInputElement;

//different syntax (don’t use in jsx)

//(<HTMLInputElement>input).value

btn.addEventListener("click", function () {

alert(input.value);

input.value = "";

});

## Handling events

* Need to tell ts the event type when calling a callback function
* Event type can also be Event

// form.addEventListener("submit", function (e) {

// e.preventDefault();

// console.log("submitted");

// });

function handleSubmit(e: SubmitEvent) {

e.preventDefault();

console.log("handle function");

}

form.addEventListener("submit", handleSubmit);

## building the todo list

**./index.html:**

<body>

<h1>Typescript Mini Project</h1>

<ul id="todolist"></ul>

<form id="todoform">

<input id="todoinput" type="text" placeholder="enter a todo" />

<button id="btn" type="submit">Click Me</button>

</form>

<script src="dist/index.js"></script>

</body>

**./src/index.ts:**

interface Todo {

text: string;

completed: boolean;

}

const todos: Todo[] = [];

const btn = document.getElementById("btn")! as HTMLButtonElement;

const input = document.getElementById("todoinput")! as HTMLInputElement;

const form = document.querySelector("form")!;

const list = document.getElementById("todolist")!;

function handleSubmit(e: SubmitEvent) {

e.preventDefault();

const newTodo: Todo = {

text: input.value,

completed: false,

};

createTodo(newTodo);

todos.push(newTodo);

input.value = "";

console.log(todos);

}

function createTodo(todo: Todo) {

const newLI = document.createElement("li");

const checkbox = document.createElement("input");

checkbox.type = "checkbox";

newLI.append(todo.text);

newLI.append(checkbox);

list.append(newLI);

}

form.addEventListener("submit", handleSubmit);

## connecting with local storage

* Can only store text or strings: JSON.stringify and JSON.parse

const todos: Todo[] = readTodos();

todos.forEach(createTodo);

function readTodos(): Todo[] {

const todosJSON = localStorage.getItem("todos");

if (todosJSON === null) {

return [];

}

return JSON.parse(todosJSON);

}

## Completed project - index.js

interface Todo {

text: string;

completed: boolean;

}

const btn = document.getElementById("btn")! as HTMLButtonElement;

const input = document.getElementById("todoinput")! as HTMLInputElement;

const form = document.querySelector("form")!;

const list = document.getElementById("todolist")!;

const todos: Todo[] = readTodos();

todos.forEach(createTodo);

function readTodos(): Todo[] {

const todosJSON = localStorage.getItem("todos");

if (todosJSON === null) { //type narrowing

return [];

}

return JSON.parse(todosJSON);

}

function saveTodos() {

localStorage.setItem("todos", JSON.stringify(todos));

}

function handleSubmit(e: SubmitEvent) {

e.preventDefault();

const newTodo: Todo = {

text: input.value,

completed: false,

};

createTodo(newTodo);

todos.push(newTodo);

localStorage.setItem("todos", JSON.stringify(todos));

input.value = "";

console.log(todos);

}

function createTodo(todo: Todo): void {

const newLI = document.createElement("li");

const checkbox = document.createElement("input");

checkbox.type = "checkbox";

checkbox.checked = todo.completed;

checkbox.addEventListener("change", function () {

todo.completed = checkbox.checked;

saveTodos();

});

newLI.append(todo.text);

newLI.append(checkbox);

list.append(newLI);

}

form.addEventListener("submit", handleSubmit);

# (JS Classes)

## The class keyword

* Classes are templates for creating objects in JS. They contain a few important pieces which allow for creation and extension of customised & well organised objects
* Allows us to create patterns for objects, what does it look like & what can it do
* Then we can instantiate instances of that class multiple times
* At its simplest:

class Player {}

//instantiate a new player

const player1 = new Player()

* Adding a method:

class Player {

taunt() {

console.log("BOO");

}

}

const player1 = new Player();

player1.taunt()

//creating a second instance

const player2 = new Player();

player2.taunt()

## Contructors

* Constructors are methods that are called when we instantiate a new instance of a class (e.g. when we create a new player)

class Player {

constructor(first, last) {

this.first = first;

this.last = last;

}

taunt() {

console.log("BOO");

}

}

const player1 = new Player("charlie", "brown");

player1.taunt();

console.log(player1.first);

console.log(player1.last);

const player2 = new Player("peter", "rabbit");

player2.taunt();

## Class fields

* Used to define fields/properties in a class
* Don’t need dynamic values to be passed in when the instance is instantiated
* Previously was done inside the constructor
* Now its above the constructor/at the top of the class and doesn’t need this keyword
* Class fields are hardcoded (cannot be dynamically set on instantiation) e.g. every player has a score of 0 and numLives of 10

class Player {

score = 0;

numLives = 10;

constructor(first, last) {

this.first = first;

this.last = last;

}

taunt() {

console.log("BOO");

}

loseLife(){

this.numLives -= 1;

}

}

## Private fields (properties)

* All fields are public, anyone can change the score
* In the past \_score was used as a convention to say don’t touch this field
* Now we have #score, that field not accessible outside of the class as it’s a private field
* If a user needs access to score, they can run getScore()
* If a user needs to set a score, they run setScore()

class Player {

#score = 0;

#numLives = 10;

constructor(first, last) {

this.first = first;

this.last = last;

this.#secret();

}

getScore() {

return this.#score;

}

setScore(newScore) {

this.#score = newScore;

}

taunt() {

console.log("boo");

}

loseLife() {

this.#numLives -= 1;

}

#secret() {

console.log("secret!");

}

}

* Private methods

## Getters

* Object accessors, syntactic sugar
* Write something that we call as if it’s a property
* It’s a method with get keyword in front of it
* We access it as if it’s a property

Class Player {

…blah blah…

get fullName() {

return `${this.first} ${this.last}`;

}

get score() {

return this.#score;

}

}

console.log(player1.score)

console.log(player1.fullname)

## Setters

* Similar syntax used for setters, a method prefixed with set keyword

class Player {

#score = 0;

#numLives = 10;

constructor(first, last) {

this.first = first;

this.last = last;

this.#secret();

}

get score() {

return this.#score;

}

set score(newScore) {

if (newScore < 0) {

throw new Error("Score must be positive!");

}

this.#score = newScore;

}

taunt() {

console.log("boo");

}

loseLife() {

this.#numLives -= 1;

}

#secret() {

console.log("secret!");

}

get fullName() {

return `${this.first} ${this.last}`;

}

set fullName(newName) {

const [first, last] = newName.split(" ");

this.first = first;

this.last = last;

}

}

const player1 = new Player("charlie", "brown");

player1.taunt();

console.log(player1.score);

player1.score = 100;

console.log(player1.score);

const player2 = new Player("peter", "rabbit");

player2.taunt();

console.log(player2.fullName);

player2.fullName = "amy adams";

console.log(player2.fullName);

## Static properties & methods

* Static fields and methods, use the static keyword in front of a class field or a method, means it exists on the class and not on an individual instance
* Static field/property:

static description = ‘Player in our game’

* can’t access player1.description
* can access Player.description
* static method:

static randomPlayer() {

return new Player(‘Andy’, ‘Samberg’)

}

* often used with a creator method or helper method

## Extending classes/inheritance

* use the extends keyword:
* class AdminPlayer extends Player {}
* Player is the base class or super class
* An instance of AdminPlayer has all the properties and methods of Player

## Super

* If there are 2 constructors (the base/parent class and the child class) only one will run, the first that js comes to which is the AdminPlayer/child constructor
* Super references/calls the constructor method of the base class

class AdminPlayer extends Player {

isAdmin = true;

constructor(first, last, powers) {

super(first, last);

this.powers = powers;

}

}

const admin = new AdminPlayer("bugs", "bunny", ["delete", "restore world"]);

# Typescript classes

## Annotating classes in typescript

* Any parameters need to be annotated
* In typescript, cant create properties in constructor with telling ts their type

class Player {

first: string;

last: string;

constructor(first: string, last: string) {

this.first = first;

this.last = last;

}

}

const charlie = new Player("Charlie", "Brown");

## class fields

* Annotate the class field (if its not annotated and the value is set, ts will infer its type)

class Player {

first: string;

last: string;

score: number = 0;

constructor(first: string, last: string) {

this.first = first;

this.last = last;

}

}

## Readonly modifier/class properties

* Put readonly keyword in front of class field
* It cannot be changed by user

class Player {

readonly first: string;

readonly last: string;

score: number = 0;

constructor(first: string, last: string) {

this.first = first;

this.last = last;

}

}

## The public modifier

* By default every property and method is public – you can access it from outside the class
* Public keyword doesn’t change default functionality, but is used for clarity to show the property is writeable from outside the class
* Underlying code/js code wont have the public keyword

class Player {

readonly first: string;

readonly last: string;

public score: number = 0;

constructor(first: string, last: string) {

this.first = first;

this.last = last;

}

}

## The private modifier

* The private keyword is ts functionality and is used to show a property or method is not accessible outside the class
* Its better to use the # (private field), as this will work (stay inaccessible) after the code compiles into js
* Cant use private keyword with #

If you use the private modifier/keyword, that property/method will still be accessible in js after the code compiles even if ts errors

class Player {

readonly firstname: string;

readonly lastname: string;

private score: number = 0;

constructor(first: string, last: string) {

this.firstname = first;

this.lastname = last;

}

}

const charlie = new Player("Charlie", "Brown");

## Parameter properties shorthand

* A shortcut for parameters being passed into the constructor
* Use the public or private keyword and you don’t need to use set the this variable in the constructor
* it compiles to the full longhand code in js

class Player {

private score: number = 0;

constructor(public first: string, public last: string) {}

}

## Getters and setters

* can have getters and setters in ts
* if ts sees a getter without a setter it will make it readonly

class Player {

constructor(

public first: string,

public last: string,

private \_score: number

) {}

private secretMethod(): void {

console.log("secret method!");

}

get fullName(): string {

return `${this.first} ${this.last}`;

}

get score(): number {

return this.\_score;

}

set score(newScore: number) {

if (newScore < 0) {

throw new Error("Score cannot be negative");

}

this.\_score = newScore;

}

}

## The protected modifier

* protected keyword makes a class field accessible from within the class and child classes only

class Player {

constructor(

public first: string,

public last: string,

protected \_score: number

) {}

private secretMethod(): void {

console.log("secret method!");

}

get fullName(): string {

return `${this.first} ${this.last}`;

}

get score(): number {

return this.\_score;

}

set score(newScore: number) {

if (newScore < 0) {

throw new Error("Score cannot be negative");

}

this.\_score = newScore;

}

}

class SuperPlayer extends Player {

public isAdmin: boolean = true;

maxScore() {

this.\_score = 9999999;

}

}

* if the field is private, a child class cannot access it

## Classes and interfaces

* interfaces can describe the shape of a class (as well as objects)

class Bike implements Colourful {

constructor(public color: string) {}

}

class Jacket implements Colourful, Printable {

constructor(public brand: string, public color: string) {}

print() {

console.log(`${this.color} ${this.brand} jacket`);

}

}

const bike1 = new Bike("red");

const jacket1 = new Jacket("Prada", "black");

## Creating abstract classes

* exists only in ts not in js
* you cannot instantiate an instance of an abstract class (with the new keyword) but you can use it as a parent/base class
* different from interfaces as the parent class can implement a lot of functionality (additional properties and methods) that an interface can’t

abstract class Employee {

constructor(public first: string, public last: string) {}

abstract getPay(): number;

greet() {

console.log("Hello!");

}

}

class FullTimeEmployee extends Employee {

constructor(first: string, last: string, private salary: number) {

super(first, last);

}

getPay(): number {

return this.salary;

}

}

class PartTimeEmployee extends Employee {

constructor(

first: string,

last: string,

private hourlyRate: number,

private hoursWorked: number

) {

super(first, last);

}

getPay(): number {

return this.hourlyRate \* this.hoursWorked;

}

}

const betty = new FullTimeEmployee("Betty", "White", 95000);

const bill = new PartTimeEmployee("Bill", "Billerson", 24, 1100);

# Generics

## Introducing generics

* generics allow us to define reusable functions and classes that work with multiple types rather than a single type
* the syntax:

function wrapInArray<T>(element: T): T[] {

return [element]

}

//note: T is shorthand convention for Type

* to call the function:

const array = wrapInArray<string>(‘hello’)

* this then types the argument to be a string and returns a string
* This is different from:

function doThing(thing: string | number): string | number {

}

* And different from:

function doThing(thing: any):any {

}

* The syntax is similar to:

const nums: Array<number> = []

const colors: Array<string> = []

## Another example of a built-in generic function

* You can specify the type of an element

const input = document.querySelector<HTMLInputElement>(‘#username’)!;

console.dir(input);

input.value = ‘hacked!’;

const btn = document.querySelector<HTMLButtonElement>(‘.btn’)!;

* Exclamation mark tells ts that the type will not be null

## Writing our first generic function

function numberIdentity(item: number):number {

return item

}

function stringIdentity(item: string):string {

return item

}

* Problem: limited to whatever types you’ve written a function for, doesn’t cover all possible types
* type ‘any’ doesn’t link the relationship between the argument and the return value

function identity<Type>(item: Type): Type {

}

interface Cat {

name: string;

breed: string;

}

//calling the function

identity<string>(‘hello’)

identity<boolean>(true)

identity<Cat>({name: ‘bluesteele’, breed: ‘siamese’})

## Writing another generic function

function getRandomElement<T>(list: T[] ): T {

const randomIndex = Math.floor(Math.random() \* list.length);

return list[randomIndex];

}

getRandomElement([4,5,6,7])

getRandomElement( [true, false, true])

getRandomElement( [{}, {}, {} ])

## Inferred generic type patterns

* typescript can infer types
* even if you don’t supply the angle bracket/type parameter, ts can infer the type from the arguments used when you call the function

## Generics, arrow functions & tsx files

* ts gets confused when we write arrow functions
* if you are working in a tsx file and are writing arrow functions, use a trailing comma after the T:

const getRandomElement = <T,>(list: T[]): T => {

const randomIndex = Math.floor(Math.random() \* list.length);

return list[randomIndex];

}

## Generics with multiple types

function merge(object1, object2) {

return {

…object1,

…object2}

* in the above, what type would we get back, ts thinks ‘any’
* in this case we can define multiple types, U is the next letter after T so convention is to use U, V, W etc)

function merge<T, U>(object1: T, object2: U) {

return {

…object1,

…object2}

* now ts says the return type is the intersection between T & U
* ts gains knowledge about the return type that is better than using any

## Adding type constraints

* we can constrain the types used in the type parameter using the extends keyword
* in the following function we are saying that the argument must be an object (it can be any type of object):

function merge<T extends object, U extends object>(object1: T, object2: U) {

return {

…object1,

…object2}

* using an interface as a constraint:

interface Lengthy{

length: number;

}

function printDoubleLength<T>(thing: T extends Lengthy): number {

return thing.length \* 2;

}

## Default type parameters

* for querySelector the default type is <Element>
* to specify a default type parameter use =

function makeEmptyList<T = number>(): T[] {

return []

}

const nums = makeEmptyArray();

const bools = makeEmptyArray<boolean>()

## Writing generic classes

interface Song {

title: string;

artist: string;

}

interface Video {

title: string;

creator: string;

resolution: string

}

class Playlist<T> {

public queue: T[] = [];

add(element: T) {

this.queue.push(element)  
}

const songs = new Playlist<Song>();

const videos = new Playlist<Video>();

videos.add({})

* all the methods of the class will use the type T
* the class can be adapted to the needs of different types

# Type Narrowing

## typeof guards

* when we have a less precise type like a union type and we have to narrow it down to a more precise type
* how would you triple a number or string depending on what input you receive?:
* answer: use a typeof guard:

function triple(value: number | string) {

if(typeof === ‘string’) {

return value.repeat(3)

}

return value \*3;

}

* useful for when working with primitive types

## truthiness guards

* truthiness type guards involve checking a value for being truthy or falsy before working with it
* this is helpful in avoiding errors when values might be null or undefined

const printLetters = (word: string | null) => {

if (!word) {

console.log(‘No word was provided’);

}else {

word.forEach(letter => console.log(letter))

}

}

* can write a truthy check and ts will know that an element is not null:

const el = document.getElementById(‘idk’)

if(el) {

//ts knows that el must be an HTMLElement

}else {

//ts knows el is null

}

* also can do the following. If you wrap an operation in a truthy check, ts will know that you can perform that operation on that particular type:

const printLetters = (word?: string) => {

if (word) {

for (let char of word) {

console.log(char)

}

}else {

console.log(‘You did not pass in a word!’)

}

}

## equality narrowing

* equality type guards involve comparing types to each other before doing certain operations with values
* by checking 2 values against one another, we can be sure they’re both the same before working with them in a type-specific way

const someFunc = (x: string | boolean, y: string | number) => {

if (x === y) {

x.toUpperCase();

y.toUpperCase();

} else {

console.log(x)

console.log(y)

}

}

* note the following. Ts doesn’t spot if you use double equals, it incorrectly thinks the type is string, whilst js can coerce a value to be a number:

const someFunc = (x: string | boolean, y: string | number) => {

if (x == y) {

y.toUpperCase();

} else {

console.log(x)

console.log(y)

}

}

someFunc(‘3’, 3)//this will error

## narrowing with the In Operator

* js’s in operator helps check if a certain property exists in an object
* this means we can use it to check if a value exists in an object according to its type alias or alisases, before working on it
* in JS:

const pet = {name: ‘Kitty, age: 20}

‘name’ in pet // true

‘breed’ in pet // false

* an example of working with the in Operator:

interface Movie {

title: string;

duration: number:

}

interface TVShow {

title: string;

numEpisodes: number;

episodeDuration: number;

}

function getRuntime(media: Movie | TVShow) {

if (‘numEpisodes’ in media) {

return media.numEpisodes \* media.episodeDuration

}

return media.duration

}

* we use the in operator when we’re working with interfaces and we need to know if a property is in that interface

## instanceof narrowing

* instanceof is a js operator that allows us to check if one thing is an instance of another (as in prototypes/classes)
* this can help narrow types when working with things like classes

const printFullDate( date: Date | string) {

if (date instanceof Date) {

return date.toUTCString();

} else {

return new Date(date).toUTCString()}

}

* anything that uses the new keyword, can use instance of:

new Array()

new Date()

[1,2,3] instanceof Array //true

* instanceof can also be used for custom classes

class User {

constructor(public username: string) {}

}

class Company {

Constructor(public name: string) {}

}

function printName(entity: User | Company) {

if(entity instanceof User) {

entity

} else {

entity

}

}

## working with type predicates

* ts allows us to write custom functions that can narrow the type of a value
* these functions have a very special return type called a type predicate
* a predicate takes the form **parameterName is Type**

interface Cat {

name: string;

numLives: number;

}

interface Dog {

name: string;

breed: string;

}

function isCat(animal: Cat | Dog): animal is Cat {

return (animal as Cat).numLives !== undefined

}

function makeNoise(animal: Cat | Dog) : string {

if(isCat(animal)) {

return ‘meow’

} else {

return ‘woof’

}

* predicate tells ts, if function returns true, we know animal is a cat

<https://www.geeksforgeeks.org/what-is-type-predicates-in-typescript/>

* Type predicates in TypeScript are functions that return a boolean value and are used to narrow down the type of a variable
* They are primarily used in conditional blocks to check whether a variable is of a particular type and then perform specific operations accordingly.
* Type predicates can be defined using the “is” keyword in TypeScript

## discriminated unions

* A common pattern in ts involves creating a literal property that is common across multiple types
* We can then narrow the type using that literal property
* In the following example, ‘kind’ is a literal property (circle or square):

interface Circle {

kind: ‘circle’;

radius: number;

}

interface Square {

kind: ‘square’;

sideLength: number;

}

* Another example. How to check if something is a Rooster, Pig or Cow if the properties are more complex or the same (can’t use ‘in’ operator to test for a property):

interface Rooster {

name: string;

weight: number;

age: number;

kind: ‘rooster’;

}

interface Cow {

name: string;

weight: number;

age: number;

kind: ‘cow’

}

interface Pig {

name: string;

weight: number;

age: number;

kind: ‘pig’;

}

Type FarmAnimal = Pig | Cow | Rooster

* We add a property that all animals will have in common. Often its called: kind, type, \_\_type

function getFarmAnimalSound(animal: FarmAnimal) {

switch(animal.kind){

case(‘pig’):

return ‘Oink’

case(‘cow’):

return ‘moo’;

case(‘rooster’):

return(‘cookadoodledoo’)

}

}

const stevie: Rooster = {

name: ‘stevie chicks’,

weight: 2,

age: 1.5

kind: ‘rooster’

}

//ts will warn us if we leave off the kind property or if we set kind to anything other than rooster

getFarmAnimalSound(stevie)// cockadoodledoo

## exhaustiveness checks with never

function getFarmAnimalSound(animal: FarmAnimal) {

switch(animal.kind){

case(‘pig’):

return ‘Oink’

case(‘cow’):

return ‘moo’;

case(‘rooster’):

return(‘cookadoodledoo’);

default:

//we should never make it here, if we handled all cases correctly

const shouldNeverGetHere: never = animal

}

}

* The default is our exhaustiveness check to make sure we handled all eventualities
* If there is an animal that is not checked for, the default will run and ts will error that that type should not be assigned to type never

# Working with Type Declarations

## Introducing type declarations

* Typescript has 2 main types of files
  + .ts files are implementation files that contain types and executable code. These are files that produce .js outputs and are where you would normally write your code
  + .d.ts files are declaration files that contain only type information (declarations of types). These files don’t produce .js outputs; they are used only for type checking
* Ts includes declaration files for all standardized built-in APIs available in JS runtimes. This includes methods and properties of built in types like string and function, names like Math and Object and their associated types
* By default, TS also includes types for things available when running inside the browser, like window and document (these are called DOM APIs)
* For ‘console’ object, its type declarations are in > lib.dom.d.ts (right click method and go to type definition). TS knows all its methods because it has a Console interface in this file
* lib.dom.d.ts is a list of interfaces and types, no executable code
* TS also knows all ‘document’ methods
* 3rd party libraries are not included in TS so need additional steps to tell TS what to do

## Using 3rd party libraries – axios

* Use jsonplaceholder api
* Make an axios request in TS
* axios comes with its own TS declaration file, which comes when you install it - index.d.ts
* in the axios package.json (in node\_modules), the ‘types’ property tells TS where to find its declaration file

import axios from "axios";

axios

.get("https://jsonplaceholder.typicode.com/users/1")

.then((res) => {

console.log("it worked", res.data);

})

.catch((err) => {

console.log("error", err);

});

## Working with axios types

* in the above example, there was no need to do extra things for ts
* however if we want to do typing, we can look at index.d.ts to see how to use types with axios

import axios from "axios";

interface User {

readonly id: number;

name: string;

username: string;

email: string;

address: {

street: string;

suite: string;

city: string;

zipcode: string;

geo: { lat: string; lng: string };

};

phone: string;

website: string;

company: {

name: string;

catchPhrase: string;

bs: string;

};

}

axios

.get<User>("https://jsonplaceholder.typicode.com/users/1")

.then((res) => {

printUser(res.data);

})

.catch((err) => {

console.log("error", err);

});

axios

.get<User[]>("https://jsonplaceholder.typicode.com/users")

.then((res) => {

res.data.forEach(printUser);

})

.catch((err) => {

console.log("error", err);

});

function printUser(user: User) {

console.log(user.name);

console.log(user.email);

console.log(user.phone);

}

* this allows us to tell TS what to anticipate and then we can enforce rules in printUser (what properties are on User object)

## Installing types separately

* lodash – methods on the \_. Object
* doesn’t come with ts declaration automatically installed – ‘could not find declaration file for module lodash’
* to install the declaration file:

npm install --save-dev @types/<package\_name>

* for example:

npm install --save-dev @types/lodash

* <https://www.typescriptlang.org/docs/handbook/declaration-files/consumption.html>
* You can then use lodash and ts will find the methods on \_

import \_ from 'lodash'

\_.partition()

* For a full list of supported npm packages refer to:
  + <https://github.com/DefinitelyTyped/DefinitelyTyped> (only shows first 1000 packages)
  + <https://yarnpkg.com/> (shows a comprehensive list)

# Modules

* Modules is a way of sharing code between files; commonjs, esmodule syntax
* TS supports modules and allows us to share code between files

## A note on namespaces

* Namespaces is the old way of sharing/organising code in TS
* TS namespaces pre-dates ES modules
* Uses the ‘namespace’ keyword
* Whilst its not deprecated, its recommended to use ES Modules instead
* You will see namespace keyword in definitelytyped declaration files

## Working without modules

* Now TS supports ES modules which uses import/export keywords
* If there is no export keyword, TS considers you are working in a script (not a module) and as such all declarations in a script are in the global namespace
* For this reason, if you write a function or a variable with the same name in 2 files, TS will error
* TS assumes you are going to handle in HTML calling the scripts in the correct order

## Using typescript modules

* Any file with an export keyword or top-level await is considered a module
* If you use the export keyword, TS will error unless you import that function/variable

/utils.ts:

export function add(x: number, y: number): number {

return x + y;

}

export function sample<T>(arr: T[]): T {

const idx = Math.floor(Math.random() \* arr.length);

return arr[idx];

}

/index.ts:

import { add, sample } from "./utils.js";

console.log(add(2, 3));

console.log(sample([1, 2, 3, 4]));

* In the /dist folder you will get the following syntax:

/index.js:

"use strict";

Object.defineProperty(exports, "\_\_esModule", { value: true });

const utils\_js\_1 = require("./utils.js");

console.log((0, utils\_js\_1.add)(2, 3));

console.log((0, utils\_js\_1.sample)([1, 2, 3, 4]));

/utils.js:

"use strict";

Object.defineProperty(exports, "\_\_esModule", { value: true });

exports.sample = exports.add = void 0;

function add(x, y) {

return x + y;

}

exports.add = add;

function sample(arr) {

const idx = Math.floor(Math.random() \* arr.length);

return arr[idx];

}

exports.sample = sample;

## Changing compilation module system

* Js in browser doesn’t know about commonjs syntax
* Could use webpack, requirejs
* If you want to use esmodules syntax in browser:
  + In the tsconfig.json:
  + "module": "ES6"
  + (if you put “” and ctrl + space bar, it will autocomplete)
  + In index.html:
  + <script type="module" src="./dist/index.js"></script>
  + Install lite-server
  + In package.json:
    - “start” : “lite-server”
  + npm start

## Import/export syntax in depth

* named exports/imports
  + export individual things
  + use {} to import

export const pi = 3.14

import {pi} from ‘./utils.js’

* default export
  + only 1 per file
  + if it’s the one main thing to export
  + export default class User
  + import <Anyname> from ‘./filename’
* can rename an export with a different name when you import it
  + import {sample as randomSample} from ‘./filename.js’

## Importing types

* importing types from different files
* export types: use export keyword in front of interface or type keyword
* normally the types disappear in js
* using ‘import type’ allows ts to work with babel:

import type {Person } from ‘./types.js’

# Webpack & Typescript

<https://webpack.js.org/>

## What’s the point of webpack

* webpack bundles all the js, css, 3rd party libraries, dependencies (100s of files) together in smallish bundles so that the browser doesn’t have too much to load
* JS is bigger now in apps; when a webpage loads it loads the HTML, CSS and 1 js file and then loads all the dependencies/scripts. Lots of scripts means lots of http requests, that can lead to a bottleneck. Coding everything in 1 JS file is unfeasible
* Webpack bundles all assets into static assets that can be loaded as a single script
* Webpack with TS requires configuration

## Setting up a project

* Project can be found at:

/Code/typescript/colt/project/03\_webpack

* When using 3rd party library, browser errors as it cannot find lodash files:

Uncaught TypeError: Failed to resolve module specifier "lodash". Relative references must start with either "/", "./", or "../".

* Also when using modules, browser has to import all the module files/dependencies. It has to load multiple files
* Webpack can help in this

## Installing webpack dependencies

* npm packages needed:
  + webpack
  + webpack-cli
  + typescript
  + ts-loader

npm install –save-dev webpack webpack-cli typescript ts-loader

* it’s a good practice to include typescript & its version in the package.json; it ensures anyone using the project gets the typescript dependency installed
* ts-loader is the middleman between ts and webpack
* they are dev dependencies

## Basic webpack config

<https://webpack.js.org/guides/typescript/#root>

* webpack.config.js:

const path = require('path');

module.exports = {

entry: './src/index.ts',

module: {

rules: [

{

test: /\.tsx?$/,

use: 'ts-loader',

exclude: /node\_modules/,

},

],

},

resolve: {

extensions: ['.tsx', '.ts', '.js'],

},

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist'),

},

};

* we can add other rules for sass and other loaders
* in package.json:

"scripts": {

"start": "lite-server",

"build": "webpack"

}

* need to remove all file extensions from imports so that webpack can resolve them
* dist/bundle.js – a compressed file
* index.html:

<script src="./dist/bundle.js"></script>

## Adding source maps

* how do we debug a bundle.js file? Source maps!
* Source maps map bundle.js code back to its pre-compiled state
* In tsconfig.json:

"sourceMap": true,

<https://webpack.js.org/guides/typescript/#source-maps>

* webpack.config.js:

devtool: 'inline-source-map',

* then:

npm run build

npm start

* your executable code should be in the devtools > sources tab> page >
* <your\_project\_name>/src
* You can then put in breakpoints and debug the code

## Fixing the missing devServer config

* Webpack.config.js should look like this:

const path = require("path");

module.exports = {

mode: "development",

entry: "./src/index.ts",

devtool: "inline-source-map",

devServer: {

static: {

directory: path.join(\_\_dirname, "./"),

},

},

module: {

rules: [

{

test: /\.tsx?$/,

use: "ts-loader",

exclude: /node\_modules/,

},

],

},

resolve: {

extensions: [".tsx", ".ts", ".js"],

},

output: {

filename: "bundle.js",

path: path.resolve(\_\_dirname, "dist"),

publicPath: "/dist",

},

};

## Webpack dev server

* Keeps the bundle in memory

npm install –save-dev webpack-dev-server

* package.json:

"scripts": {

"start": "lite-server",

"build": "webpack",

"serve": "webpack serve"

},

* then run webpack dev server

npm run serve

* it will log which port the server is running on

<http://localhost:8080>

* bundle file in /dist is uncompiled

## Production configuration

* if mode is development, there’s no minification
* want to be able to run dev server and have production configuration when you run npm run build
* a common approach is to have a dev webpack.config.js and a production one
* copy contents of webpack.config.js to webpack.prod.js and set mode to production
* in package.json:

“build”: “webpack –config webpack.prod.js”

* then run:

npm run build

* development mode is accessed using npm run serve, production mode is accessed using npm run build
* output: {
* filename: [contenthash].bundle.js
* will give a hashed filename to the bundle filename
* helps the browser identify the bundle has changed
* this creates additional files so install clean-webpack-plugin

npm install –save-dev clean-webpack-plugin

* in webpack.prod.js:

const { CleanWebpackPlugin } = require('clean-webpack-plugin');

plugins: [

new CleanWebpackPlugin(),

],

* note: problem with contenthash, is the index.html will not know the name of the file!
* Rename webpack.config.js to webpack.dev.js and add to package.json:

"serve": "webpack-dev-server --config webpack.dev.js"

## colt’s webpack course

<https://www.youtube.com/watch?v=3On5Z0gjf4U&list=PLblA84xge2_zwxh3XJqy6UVxS60YdusY8>

<https://github.com/Colt/webpack-demo-app>

# React & Typescript

## Introduction

* ~~Create react app integrates with ts~~
* Why use Typescript with React (or in general?)
  + It leads to better js code because you can catch errors earlier
  + Easy to miss a prop or pass the wrong kind of data to a prop
  + You can see those errors earlier and fix them immediately
  + Leads to more complex code as defining types can be tricky

## Vite + React + Typescript

<https://blog.logrocket.com/build-react-typescript-app-vite/>

* npm create vite@latest
* When prompted enter:
  + Project name
  + React as the framework
  + Typescript as the variant
* Then run:
  + cd into project directory
  + npm install
  + npm run dev

## Create react app with typescript

npx create-react-app my-app –template typescript

* (react typescript files use .tsx extension)
* or you can install typescript to an existing project or a project that’s not using cra:

npm install –save typescript @types/node @types/react @types/react-dom @types/jest

## Our first component

* react-typescript cheatsheets:

<https://github.com/typescript-cheatsheets/react>

<https://react-typescript-cheatsheet.netlify.app/docs/basic/setup>

* note: not the official docs, but a good resource to get started

## A note on React functional components

* React.FunctionComponent is discouraged. In older code bases you will find:

const Greeter: React.FC = () => {

return <h1>Hello!</h1>

}

* Better syntax is:

function Greeter(): JSX.Element {

return <h1>Hello!!!</h1>;

}

export default Greeter;

* Note: since TS 5.1 and React 18, React.FC is now ok to use, it is not deprecated (contrary to the above previous note)

## Props with typescript

* When you give a component a prop, it has this error message:

Type '{ person: string; }' is not assignable to type 'IntrinsicAttributes'.

Property 'person' does not exist on type 'IntrinsicAttributes'

* Basically it means the component wasn’t expecting that prop

function Greeter(props: { person: string }): JSX.Element {

return <h1>Hello {props.person}!!!</h1>;

}

* In bigger projects, don’t put the types inline:

interface GreeterProps {

person: string;

}

function Greeter(props: GreeterProps): JSX.Element {

return <h1>Hello {props.person}!!!</h1>;

}

* Also its common to omit the props object and destructure from props:

function Greeter({person}: GreeterProps): JSX.Element {

return <h1>Hello {person}!!!</h1>;

}

## Creating our ShoppingList Component

* ShoppingList Component:

/src/components/ShoppingList.tsx

import React from "react";

interface Item {

id: number;

product: string;

quantity: number;

}

interface ShoppingListProps {

items: Item[];

}

export default function ShoppingList({

items,

}: ShoppingListProps): JSX.Element {

return (

<div>

<h1>Shopping List</h1>

<ul>

{items.map((item) => {

return (

<li key={item.id}>

{item.product} - {item.quantity}

</li>

);

})}

</ul>

</div>

);

}

/src/App.tsx:

import React from "react";

import "./App.css";

import ShoppingList from "./components/ShoppingList";

function App() {

const items = [

{ id: 1, product: "Lemon", quantity: 3 },

{ id: 2, product: "Chicken", quantity: 2 },

];

return (

<>

<ShoppingList items={items} />

</>

);

}

export default App;

## useState with TypeScript

/src/App.tsx:

import React, { useState } from "react";

import "./App.css";

import ShoppingList from "./components/ShoppingList";

import Item from "./models/item";

function App() {

const [items, setItems] = useState<Item[]>([]);

return (

<>

<ShoppingList items={items} />

</>

);

}

export default App;

/src/models/item.ts:

export default interface Item {

id: number;

product: string;

quantity: number;

}

## creating our ShoppingItem Form

import React from "react";

function ShoppingListForm(): JSX.Element {

function handleSubmit(e: React.FormEvent) {

e.preventDefault();

console.log("submitted!");

}

return (

<form onSubmit={handleSubmit}>

<input type="text" placeholder="Product Name" />

<button type="submit">Add Item</button>

</form>

);

}

export default ShoppingListForm;

## useRef with Typescript

/src/components/ShoppingListForm.tsx:

import React, { useRef } from "react";

interface ShoppingListFormProps {

onAddItem: (item: string) => void;

}

function ShoppingListForm({ onAddItem }: ShoppingListFormProps): JSX.Element {

const inputRef = useRef<HTMLInputElement>(null);

function handleSubmit(e: React.FormEvent) {

e.preventDefault();

console.log("submitted!");

const newProduct = inputRef.current!.value;

onAddItem(newProduct);

inputRef.current!.value = "";

}

return (

<form onSubmit={handleSubmit}>

<input type="text" placeholder="Product Name" ref={inputRef} />

<button type="submit">Add Item</button>

</form>

);

}

export default ShoppingListForm;

/src/App.tsx:

import React, { useState } from "react";

import "./App.css";

import ShoppingList from "./components/ShoppingList";

import ShoppingListForm from "./components/ShoppingListForm";

import Item from "./models/item";

function App() {

const [items, setItems] = useState<Item[]>([]);

const addItem = (product: string) => {

console.log("made it to app component");

console.log(product);

};

return (

<>

<ShoppingList items={items} />

<ShoppingListForm onAddItem={addItem} />

</>

);

}

export default App;

## updating the state

* To get unique IDs:

npm install uuid

npm install –save-dev @types/uuid

* To update the state:

import React, { useState } from "react";

import "./App.css";

import ShoppingList from "./components/ShoppingList";

import ShoppingListForm from "./components/ShoppingListForm";

import Item from "./models/item";

import { v4 as getId } from "uuid";

function App() {

const [items, setItems] = useState<Item[]>([]);

const addItem = (product: string) => {

setItems([...items, { id: getId(), product, quantity: 1 }]);

};

return (

<>

<ShoppingList items={items} />

<ShoppingListForm onAddItem={addItem} />

</>

);

}

export default App;

## finishing touches

* Adding quantity field:

import React, { useRef } from "react";

interface ShoppingListFormProps {

onAddItem: (item: string, quantity: number) => void;

}

function ShoppingListForm({ onAddItem }: ShoppingListFormProps): JSX.Element {

const productInputRef = useRef<HTMLInputElement>(null);

const quantityInputRef = useRef<HTMLInputElement>(null);

function handleSubmit(e: React.FormEvent) {

e.preventDefault();

console.log("submitted!");

const newProduct = productInputRef.current!.value;

const quantity = parseInt(quantityInputRef.current!.value);

onAddItem(newProduct, quantity);

productInputRef.current!.value = "";

quantityInputRef.current!.value = "1";

productInputRef.current?.focus();

}

return (

<form onSubmit={handleSubmit}>

<input type="text" placeholder="Product Name" ref={productInputRef} />

<input type="number" min={0} ref={quantityInputRef} />

<button type="submit">Add Item</button>

</form>

);

}

export default ShoppingListForm;

in /src/App.tsx:

const addItem = (product: string, quantity: number) => {

console.log("made it to app component");

console.log(product);

setItems([...items, { id: getId(), product, quantity }]);

};

* Normally colt would use useState in forms

## React + Typescript wrapup

* Use tsx files
* Mostly types come up with props
* Hooks are generics so you need to pass in a type (of the data that will be held by it)

# Typescript + React

(Taken from Stephen Grider – Typescript the complete developers guide)

(tutorial project in local disk/code/typescript/redux\_ts/stephen/react-ts-tutorial/)

## Introduction

* To install Vite + React + Typescript:

1. npm create vite@latest
2. Name the project
3. Choose react
4. Choose Typescript
5. cd into project directory
6. npm install
7. npm run dev

* React + ts files have .tsx extension

## Child props

* We need to define an interface in the child to define what props child expects to receive
* 2 big checks by typescript:
  + Are we providing the correct props to child when we show it in Parent?
  + Are we using the correctly named & typed props in Child?

interface ChildProps {

color: string;

}

export const Child = ({color}: ChildProps) => {

return <div>{color}</div>;

};

* If we don’t put the color prop in Parent, Parent will show an error

## 3 different approaches to applying the interface

* 1) first approach:
  + export const Child = ({color}: ChildProps) => {
  + but at the moment, ts doesn’t understand that this is a React component
  + React components can optionally provide these properties:
    - propTypes
    - displayName
    - defaultProps
    - contextTypes
  + ts doesn’t know that we’re making a React component, so it thinks that Child will not have these properties
  + ts will error if we try to set Child.displayName
* 2) a way to fix this:
  + export const ChildAsFC: React.FC<ChildProps> = ({ color }) => {
  + return <div>{color}</div>;
  + };
  + If we write ChildAsFC. Autocomplete finds the list of properties on the component, including displayName. There is no ts error if we use this property

## Property 'children' does not exist

* In the upcoming lecture, we will be making use of the children prop in our Child and ChildAsFC components. You will eventually run into the following error:

TS2339: Property 'children' does not exist on type 'ChildProps'.

* React 18 introduces a breaking change with the removal of implicit children in React.FunctionComponent types.
* To fix this, we now must include the children in the interface along with our other props:

interface ChildProps {

color: string;

onClick: () => void;

children?: React.ReactNode;

}

## How to pass a function from parent to child

src/props/Child:

interface ChildProps {

color: string;

handleClick: () => void;

}

export const Child = ({ color, handleClick }: ChildProps) => {

return (

<div>

{color}

<button onClick={handleClick}>Click Me</button>

</div>

);

};

export const ChildAsFC: React.FC<ChildProps> = ({ color, handleClick }) => {

return (

<div>

{color}

<button onClick={handleClick}>Click Me</button>

</div>

);

};

Src/props/Parent

import { Child, ChildAsFC } from "./Child";

const Parent = () => {

return (

<>

<Child color="red" handleClick={() => alert("hello")} />;

<ChildAsFC color="blue" handleClick={() => alert("hello from FC")} />

</>

);

};

export default Parent;

## Annotations with children

* When using children prop (in the parent, the jsx that is included between the opening and closing brackets), both approach 1 and approach 2 will error if children prop is not provided to the interface
* You need to annotate the children prop, before using it:

interface ChildProps {

color: string;

onClick: () => void;

children?: React.ReactNode;

}

## State with typescript

import { useState } from "react";

const GuestList: React.FC = () => {

const [name, setName] = useState("");

return (

<div>

<h3>Guest List</h3>

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

<button>Add Guest</button>

</div>

);

};

export default GuestList;

* In the above code, so far, apart from React.FC, have not needed to put in any additional info for ts

## Type inference with state

* We provide a default value when we use useState
* Ts infers the type of the state variable from the default value
* If we supply an empty array, ts assumes the type never (an array that’s always empty)
* We need to tell ts, what type the state should be

### Example 1

import React, { useState } from "react";

const GuestList: React.FC = () => {

const [name, setName] = useState("");

const [guests, setGuests] = useState<string[]>([]);

const handleClick = (e: React.FormEvent) => {

e.preventDefault();

setGuests([...guests, name]);

setName("");

};

return (

<div>

<h3>Guest List</h3>

<div>

<form action="">

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

<button type="submit" onClick={handleClick}>

Add Guest

</button>

</form>

</div>

<div>

List of Guests

<ul>

{guests.map((guest) => {

return <li key={guest}>{guest}</li>;

})}

</ul>

</div>

</div>

);

};

export default GuestList;

### Example 2

import React, { useState } from "react";

const users = [

{ name: "Sarah", age: 20 },

{ name: "Alex", age: 30 },

{ name: "Bert", age: 40 },

{ name: "Bob", age: 10 },

{ name: "Terry", age: 110 },

];

const UserSearch: React.FC = () => {

const [name, setName] = useState("");

const [user, setUser] = useState<{ name: string; age: number } | undefined>();

function handleClick() {

const foundUser = users.find((user) => {

return user.name === name;

});

setUser(foundUser);

}

return (

<div>

UserSearch

<input

value={name}

onChange={(e) => setName(e.target.value)}

type="text"

/>

<button onClick={handleClick}>Find User</button>

<div>{user?.name}</div>

<div>{user && user.age}</div>

</div>

);

};

export default UserSearch;

* We need to render user conditionally as it could be undefined (otherwise ts will give an error)

## Typing standalone event handlers

* If event is handled inline, ts does not error
* However if you pass a separate function to the onChange handler, then ts will error when you pass the event object to the standalone change handler function, as ts will infer the any type to the event object, it will not infer the HTMLInputElement properties to it
* To remedy this we need to apply a type to the event object:

const EventComponent: React.FC = () => {

function handleChange(e: React.ChangeEvent<HTMLInputElement>) {

console.log(e);

}

return (

<div>

<input type="text" onChange={handleChange} />

<input type="text" onChange={(e) => console.log("hello", e)} />

</div>

);

};

export default EventComponent;

* Change event is only applicable to text input, radio button, check boxes, text area etc

## Other type of events -drag, form submission events

* Drag events: to check the event type, in VSCode, mouse over ‘onDragStart’, this will show: React.DragEventHandler<HTMLDivElement>, you will need to omit ‘Handler’ from this:

function handleDragStart(e: React.DragEvent<HTMLDivElement>) {

console.log("im being dragged", e);

}

return

<div draggable onDragStart={handleDragStart}>

Drag Me!

</div>

* If you Ctrl + click on DragEvent or ChangeEvent, you will be taken to the types file which will show you all event types
* <https://react.dev/reference/react-dom/components/common> shows you all the events available on React common components

## Applying types to Refs

const inputRef = useRef<HTMLInputElement>();

* Ctrl + click HTMLInputElement to get a full list of HTMLElement types
* To avoid ts errors we need to tell ts, that inputRef could be pointed to null and its initial value is null (ts thinks that inputRef might not be pointing to an element):

const inputRef = useRef<HTMLInputElement | null>(null);

return

<input

ref={inputRef}

value={name}

onChange={(e) => setName(e.target.value)}

type="text"

/>

* To focus the input on mount:

useEffect(() => {

if (!inputRef.current) {

return;

}

inputRef.current.focus();

}, []);

* You can use any as an escape hatch:

const inputRef = useRef<any>();

# Typescript + Redux

(Taken from Stephen Grider – Typescript the complete developers guide)

## Installation

* To set up a project:

1. Install vite + react + typescript:
   1. Npm create vite@latest
   2. Choose project name
   3. Select React
   4. Select Typescript
   5. Cd into project directory
   6. Npm install
2. Install redux, react-redux and axios:
   1. npm install --save-exact react-redux redux @types/react-redux redux-thunk axios

## searching npm

registry.npmjs.org/-/v1/search?text=react

## file structure

/components

App.tsx

RepositoriesList.tsx

/store

index.ts

/store/reducers

/store/actionCreators

/store/middlewares

## Applying typescript to the reducer – annotating the return type

//first create js code

//then add ts

interface RepositoriesState {

loading: boolean;

error: string | null;

data: string[];

}

const reducer = (state: RepositoriesState, action: any): RepositoriesState => {

switch (action.type) {

case "search\_repositories":

return { loading: true, error: null, data: [] };

case "search\_repositories\_success":

return { loading: false, error: null, data: action.payload };

case "search\_repositories\_error":

return { loading: false, error: action.payload, data: [] };

default:

return state;

}

};

export default reducer;

* this shows that the reducer function must always return the RepositoriesState type

## typing an action

* change action from any to a specific type

//first create js code

//then add ts

interface RepositoriesState {

loading: boolean;

error: string | null;

data: string[];

}

interface Action {

type: string;

payload?: any;

}

const reducer = (

state: RepositoriesState,

action: Action

): RepositoriesState => {

switch (action.type) {

case "search\_repositories":

return { loading: true, error: null, data: [] };

case "search\_repositories\_success":

return { loading: false, error: null, data: action.payload };

case "search\_repositories\_error":

return { loading: false, error: action.payload, data: [] };

default:

return state;

}

};

export default reducer;

* problem: payload is set to any

## separate interfaces for actions

* create an interface for each action: search action, success action, error action

//first create js code

//then add ts

interface RepositoriesState {

loading: boolean;

error: string | null;

data: string[];

}

interface SearchRepositoriesAction {

type: "search\_repositories";

}

interface SearchRepositoriesSuccessAction {

type: "search\_repositories\_success";

payload: string[];

}

interface SearchRepositoriesErrorAction {

type: "search\_repositories\_error";

payload: string;

}

const reducer = (

state: RepositoriesState,

action:

| SearchRepositoriesAction

| SearchRepositoriesSuccessAction

| SearchRepositoriesErrorAction

): RepositoriesState => {

switch (action.type) {

case "search\_repositories":

return { loading: true, error: null, data: [] };

case "search\_repositories\_success":

return { loading: false, error: null, data: action.payload };

case "search\_repositories\_error":

return { loading: false, error: action.payload, data: [] };

default:

return state;

}

};

export default reducer;

* whenever we put in the action argument, we specify the types of action that it could be, using the action interfaces
* switch statements function as a type guard in ts; they check which case is applicable so ts knows what the action object should look like

## Better way to type actions

//first create js code

//then add ts

interface RepositoriesState {

loading: boolean;

error: string | null;

data: string[];

}

// interface Action {

// type: string;

// payload?: any;

// }

interface SearchRepositoriesAction {

type: "search\_repositories";

}

interface SearchRepositoriesSuccessAction {

type: "search\_repositories\_success";

payload: string[];

}

interface SearchRepositoriesErrorAction {

type: "search\_repositories\_error";

payload: string;

}

type Action =

| SearchRepositoriesAction

| SearchRepositoriesSuccessAction

| SearchRepositoriesErrorAction;

const reducer = (

state: RepositoriesState,

action: Action

): RepositoriesState => {

switch (action.type) {

case "search\_repositories":

return { loading: true, error: null, data: [] };

case "search\_repositories\_success":

return { loading: false, error: null, data: action.payload };

case "search\_repositories\_error":

return { loading: false, error: action.payload, data: [] };

default:

return state;

}

};

export default reducer;

## adding an action type enum

* to avoid duplication of strings use an enum

//first create js code

//then add ts

interface RepositoriesState {

loading: boolean;

error: string | null;

data: string[];

}

// interface Action {

// type: string;

// payload?: any;

// }

interface SearchRepositoriesAction {

type: ActionType.SEARCH\_REPOSITORIES;

}

interface SearchRepositoriesSuccessAction {

type: ActionType.SEARCH\_REPOSITORIES\_SUCCESS;

payload: string[];

}

interface SearchRepositoriesErrorAction {

type: ActionType.SEARCH\_REPOSITORIES\_ERROR;

payload: string;

}

type Action =

| SearchRepositoriesAction

| SearchRepositoriesSuccessAction

| SearchRepositoriesErrorAction;

enum ActionType {

SEARCH\_REPOSITORIES = "search\_repositories",

SEARCH\_REPOSITORIES\_SUCCESS = "search\_repositories\_success",

SEARCH\_REPOSITORIES\_ERROR = "search\_repositories\_error",

}

const reducer = (

state: RepositoriesState,

action: Action

): RepositoriesState => {

switch (action.type) {

case ActionType.SEARCH\_REPOSITORIES:

return { loading: true, error: null, data: [] };

case ActionType.SEARCH\_REPOSITORIES\_SUCCESS:

return { loading: false, error: null, data: action.payload };

case ActionType.SEARCH\_REPOSITORIES\_ERROR:

return { loading: false, error: action.payload, data: [] };

default:

return state;

}

};

export default reducer;

## Refactoring into different files & directories

* project now has this structure:

**/src/state/actions/index.ts:**

import { ActionType } from "../action-types";

interface SearchRepositoriesAction {

type: ActionType.SEARCH\_REPOSITORIES;

}

interface SearchRepositoriesSuccessAction {

type: ActionType.SEARCH\_REPOSITORIES\_SUCCESS;

payload: string[];

}

interface SearchRepositoriesErrorAction {

type: ActionType.SEARCH\_REPOSITORIES\_ERROR;

payload: string;

}

export type Action =

| SearchRepositoriesAction

| SearchRepositoriesSuccessAction

| SearchRepositoriesErrorAction;

**/src/state/action-types/index.ts:**

export enum ActionType {

SEARCH\_REPOSITORIES = "search\_repositories",

SEARCH\_REPOSITORIES\_SUCCESS = "search\_repositories\_success",

SEARCH\_REPOSITORIES\_ERROR = "search\_repositories\_error",

}

**/src/state/reducers/respositoriesReducer.ts:**

//first create js code

//then add ts

import { ActionType } from "../action-types";

import { Action } from "../actions";

interface RepositoriesState {

loading: boolean;

error: string | null;

data: string[];

}

// interface Action {

// type: string;

// payload?: any;

// }

const initialState = {

loading: false,

error: null,

data: [],

}

const reducer = (

state: RepositoriesState = initialState,

action: Action

): RepositoriesState => {

switch (action.type) {

case ActionType.SEARCH\_REPOSITORIES:

return { loading: true, error: null, data: [] };

case ActionType.SEARCH\_REPOSITORIES\_SUCCESS:

return { loading: false, error: null, data: action.payload };

case ActionType.SEARCH\_REPOSITORIES\_ERROR:

return { loading: false, error: action.payload, data: [] };

default:

return state;

}

};

export default reducer;

* if the project gets bigger, then in the actions directory individual files for the different action groups could be created, e.g. repositoriesActions.ts, usersActions.ts and then imported into the index.ts

## annotating try catch block/ adding action creators & request logic

import axios from "axios";

import { ActionType } from "../action-types";

import { Dispatch } from "redux";

import { Action } from "../actions";

export const searchRepositories = (term: string) => {

return async (dispatch: Dispatch<Action>) => {

dispatch({ type: ActionType.SEARCH\_REPOSITORIES });

try {

const { data } = await axios.get(

"https://registry.npmjs.org/-/v1/search",

{ params: { text: term } }

);

const names = data.objects.map((result: any) => {

return result.package.name;

});

dispatch({

type: ActionType.SEARCH\_REPOSITORIES\_SUCCESS,

payload: names,

});

} catch (err: any) {

dispatch({

type: ActionType.SEARCH\_REPOSITORIES\_ERROR,

payload: err.message,

});

}

};

};

* if we don’t annotate the dispatch argument, dispatch will accept any payload or action type instead of those that are specified in actions

## setting up exports

* this is standard redux setup:

**/state/reducers/index.ts:**

import { combineReducers } from "redux";

import repositoriesReducer from "./repositoriesReducer";

const reducers = combineReducers({

repositories: repositoriesReducer,

});

export default reducers;

**/src/state/store.ts:**

import { createStore, applyMiddleware } from "redux";

import { thunk } from "redux-thunk";

import reducers from "./reducers";

export const store = createStore(reducers, {}, applyMiddleware(thunk));

**/src/state/index.ts:**

export \* from "./store";

export \* as actionCreators from "./action-creators";

## wiring up to React

/src/components/App.tsx:

import "./App.css";

import { Provider } from "react-redux";

import { store } from "../state";

import RepositoriesList from "./RepositoriesList";

function App() {

return (

<Provider store={store}>

<div>

<h1>Redux TypeScript Tutorial</h1>

<hr />

<h3>Search for a package</h3>

<RepositoriesList />

</div>

</Provider>

);

}

export default App;

/src/components/RepositoriesList.tsx:

import React, { useState } from "react";

const RepositoriesList: React.FC = () => {

const [term, setTerm] = useState("");

function handleSubmit(e: React.FormEvent<HTMLFormElement>) {

e.preventDefault();

}

return (

<div>

<form onSubmit={handleSubmit}>

<input

type="text"

value={term}

onChange={(e) => setTerm(e.target.value)}

/>

<button>Search</button>

</form>

</div>

);

};

export default RepositoriesList;

## calling an action creator

/src/state/RepositoriesList.tsx:

import React, { useState } from "react";

import { useDispatch } from "react-redux";

import { actionCreators } from "../state";

const RepositoriesList: React.FC = () => {

const [term, setTerm] = useState("");

const dispatch = useDispatch()

function handleSubmit(e: React.FormEvent<HTMLFormElement>) {

e.preventDefault();

dispatch(actionCreators.searchRepositories(term) as any)

}

return (

<div>

<form onSubmit={handleSubmit}>

<input

type="text"

value={term}

onChange={(e) => setTerm(e.target.value)}

/>

<button>Search</button>

</form>

</div>

);

};

export default RepositoriesList;

## setting up a useActions hook to simplify the dispatch call

/src/hooks/useActions.ts:

import { useDispatch } from "react-redux";

import { bindActionCreators } from "redux";

import { actionCreators } from "../state";

export const useActions = () => {

const dispatch = useDispatch();

return bindActionCreators(actionCreators, dispatch);

//bindActionCreators returns an object that has the following structure:

//{ searchRepositories: dispatch(searchRepositories),

//otherAction: dispatch(otherAction)}

};

## Using the useActions hook

import React, { useState } from "react";

import { useActions } from "../hooks/useActions";

const RepositoriesList: React.FC = () => {

const [term, setTerm] = useState("");

const { searchRepositories } = useActions();

function handleSubmit(e: React.FormEvent<HTMLFormElement>) {

e.preventDefault();

searchRepositories(term);

}

return (

<div>

<form onSubmit={handleSubmit}>

<input

type="text"

value={term}

onChange={(e) => setTerm(e.target.value)}

/>

<button>Search</button>

</form>

</div>

);

};

export default RepositoriesList;

## useSelector, React-redux and typing

<https://react-redux.js.org/tutorials/typescript-quick-start>

<https://react-redux.js.org/using-react-redux/usage-with-typescript>

* react-redux doesn’t know about the types in the store
* so we have to write some extra code when using useSelector

**/src/state/reducers/index.ts:**

import { combineReducers } from "redux";

import repositoriesReducer from "./repositoriesReducer";

const reducers = combineReducers({

repositories: repositoriesReducer,

});

export default reducers;

//creating a root type to tell ts/react-redux what type of data is coming out of the store

export type RootState = ReturnType<typeof reducers>;

**/src/state/index.ts:**

export \* from "./store";

export \* as actionCreators from "./action-creators";

export \* from "./reducers";

## setting up and using useTypedSelectorHook

**/src/hooks/useTypedSelector.ts:**

import { useSelector, TypedUseSelectorHook } from "react-redux";

import { RootState } from "../state";

export const useTypedSelector: TypedUseSelectorHook<RootState> = useSelector;

**/src/components/RepositoriesList.tsx:**

import React, { useState } from "react";

import { useTypedSelector } from "../hooks/useTypedSelector";

import { useActions } from "../hooks/useActions";

const RepositoriesList: React.FC = () => {

const [term, setTerm] = useState("");

const { searchRepositories } = useActions();

const { error, loading, data } = useTypedSelector(

(state) => state.repositories

);

console.log(data);

function handleSubmit(e: React.FormEvent<HTMLFormElement>) {

e.preventDefault();

searchRepositories(term);

}

return (

<div>

<form onSubmit={handleSubmit}>

<input

type="text"

value={term}

onChange={(e) => setTerm(e.target.value)}

/>

<button>Search</button>

</form>

<hr />

{error && <div>Something went wrong: {error}</div>}

{loading && <div>Loading...</div>}

<div>

{data &&

data.map((item) => {

return <div key={item}>{item}</div>;

})}

</div>

</div>

);

};

export default RepositoriesList;

## Big issues with Redux/React-Redux + Typescript

* imports between files can turn into a mess very quickly
  + so all imports were done from state/index.ts
* communicating types over to your components can be challenging
  + set up RootState in reducers/index.ts
  + set up useTypedSelector hook
* type definition files for Redux, React-Redux and others are possibly over-engineered
  + you can look at the type definition files for redux, react-redux, redux-thunk
  + you can use any

# Typescript + Redux Toolkit

(Taken from Redux Toolkit docs)

<https://redux-toolkit.js.org/tutorials/quick-start>

<https://redux-toolkit.js.org/tutorials/typescript>

## Installation

* Install vite + react +typescript
* Install redux Toolkit and React-Redux & react-redux types:

npm install @reduxjs/toolkit react-redux @types/react-redux

## Part 1 – Set up Redux Toolkit

## Create a redux store

/src/app/store.ts:

import { configureStore } from '@reduxjs/toolkit'

export const store = configureStore({

reducer: {},

})

// Infer the `RootState` and `AppDispatch` types from the store itself

export type RootState = ReturnType<typeof store.getState>

// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}

export type AppDispatch = typeof store.dispatch

* This creates a Redux store, and also automatically configure the Redux DevTools extension so that you can inspect the store while developing.

## Provide the redux store to React

/src/main.tsx:

import React from 'react'

import ReactDOM from 'react-dom'

import './index.css'

import App from './App'

import { store } from './app/store'

import { Provider } from 'react-redux'

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

## Create a redux state slice

/src/features/counter/counterSlice.ts:

import { createSlice } from '@reduxjs/toolkit'

import type { PayloadAction } from '@reduxjs/toolkit'

export interface CounterState {

value: number

}

const initialState: CounterState = {

value: 0,

}

export const counterSlice = createSlice({

name: 'counter',

initialState,

reducers: {

increment: (state) => {

// Redux Toolkit allows us to write "mutating" logic in reducers. It

// doesn't actually mutate the state because it uses the Immer library,

// which detects changes to a "draft state" and produces a brand new

// immutable state based off those changes

state.value += 1

},

decrement: (state) => {

state.value -= 1

},

incrementByAmount: (state, action: PayloadAction<number>) => {

state.value += action.payload

},

},

})

// Action creators are generated for each case reducer function

export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer

## Add slice reducers to the store

/src/app/store.ts:

import { configureStore } from '@reduxjs/toolkit'

import counterReducer from '../features/counter/counterSlice'

export const store = configureStore({

reducer: {

counter: counterReducer,

},

})

// Infer the `RootState` and `AppDispatch` types from the store itself

export type RootState = ReturnType<typeof store.getState>

// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}

export type AppDispatch = typeof store.dispatch

## use redux state and actions in React components

/src/features/counter/Counter.tsx:

import React from 'react'

import type { RootState } from '../../app/store'

import { useSelector, useDispatch } from 'react-redux'

import { decrement, increment } from './counterSlice'

export function Counter() {

const count = useSelector((state: RootState) => state.counter.value)

const dispatch = useDispatch()

return (

<div>

<div>

<button

aria-label="Increment value"

onClick={() => dispatch(increment())}

>

Increment

</button>

<span>{count}</span>

<button

aria-label="Decrement value"

onClick={() => dispatch(decrement())}

>

Decrement

</button>

</div>

</div>

)

}

## Part 2 – Add typescript

## Define Root State and Dispatch Types

* Using configureStore should not need any additional typings. You will, however, want to extract the RootState type and the Dispatch type so that they can be referenced as needed. Inferring these types from the store itself means that they correctly update as you add more state slices or modify middleware settings
* Since those are types, it's safe to export them directly from your store setup file such as app/store.ts and import them directly into other files.

import { configureStore } from '@reduxjs/toolkit'

// ...

export const store = configureStore({

reducer: {

posts: postsReducer,

comments: commentsReducer,

users: usersReducer,

},

})

// Infer the `RootState` and `AppDispatch` types from the store itself

export type RootState = ReturnType<typeof store.getState>

// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}

export type AppDispatch = typeof store.dispatch

## Define typed hooks

* While it's possible to import the RootState and AppDispatch types into each component, it's better to create typed versions of the useDispatch and useSelector hooks for usage in your application. This is important for a couple reasons:
  + For useSelector, it saves you the need to type (state: RootState) every time
  + For useDispatch, the default Dispatch type does not know about thunks. In order to correctly dispatch thunks, you need to use the specific customized AppDispatch type from the store that includes the thunk middleware types, and use that with useDispatch. Adding a pre-typed useDispatch hook keeps you from forgetting to import AppDispatch where it's needed.
* Since these are actual variables, not types, it's important to define them in a separate file such as app/hooks.ts, not the store setup file. This allows you to import them into any component file that needs to use the hooks, and avoids potential circular import dependency issues.

import { useDispatch, useSelector } from 'react-redux'

import type { RootState, AppDispatch } from './store'

// Use throughout your app instead of plain `useDispatch` and `useSelector`

export const useAppDispatch = useDispatch.withTypes<AppDispatch>()

export const useAppSelector = useSelector.withTypes<RootState>()

## Application usage – define slice state and action types

* Each slice file should define a type for its initial state value, so that createSlice can correctly infer the type of state in each case reducer.
* All generated actions should be defined using the PayloadAction<T> type from Redux Toolkit, which takes the type of the action.payload field as its generic argument
* You can safely import the RootState type from the store file here. It's a circular import, but the TypeScript compiler can correctly handle that for types. This may be needed for use cases like writing selector functions

import { createSlice } from '@reduxjs/toolkit'

import type { PayloadAction } from '@reduxjs/toolkit'

import type { RootState } from '../../app/store'

// Define a type for the slice state

interface CounterState {

value: number

}

// Define the initial state using that type

const initialState: CounterState = {

value: 0,

}

export const counterSlice = createSlice({

name: 'counter',

// `createSlice` will infer the state type from the `initialState` argument

initialState,

reducers: {

increment: (state) => {

state.value += 1

},

decrement: (state) => {

state.value -= 1

},

// Use the PayloadAction type to declare the contents of `action.payload`

incrementByAmount: (state, action: PayloadAction<number>) => {

state.value += action.payload

},

},

})

export const { increment, decrement, incrementByAmount } = counterSlice.actions

// Other code such as selectors can use the imported `RootState` type

export const selectCount = (state: RootState) => state.counter.value

export default counterSlice.reducer

* The generated action creators will be correctly typed to accept a payload argument based on the PayloadAction<T> type you provided for the reducer. For example, incrementByAmount requires a number as its argument.
* In some cases, TypeScript may unnecessarily tighten the type of the initial state. If that happens, you can work around it by casting the initial state using as, instead of declaring the type of the variable:

// Workaround: cast state instead of declaring variable type

const initialState = {

value: 0,

} satisfies CounterState as CounterState

## Use typed hooks in components

/src/counter/Counter.tsx:

import React, { useState } from 'react'

import { useAppSelector, useAppDispatch } from 'app/hooks'

import { decrement, increment } from './counterSlice'

export function Counter() {

// The `state` arg is correctly typed as `RootState` already

const count = useAppSelector((state) => state.counter.value)

const dispatch = useAppDispatch()

// omit rendering logic

}

## Further Details

<https://redux-toolkit.js.org/usage/usage-with-typescript>

# ChatGPT Explanation: Typescript + React + Redux Toolkit

* Integrating TypeScript into a React application that uses Redux Toolkit involves setting up TypeScript for both React components and Redux slices. Redux Toolkit itself is designed to work seamlessly with TypeScript, so the integration is relatively straightforward. Let's walk through the steps with examples:
* (Typescript annotations in yellow)

## Step 1: Setting Up a TypeScript React Application

* First, create a new React application using Create React App with TypeScript:

```bash

npx create-react-app my-app --template typescript

cd my-app

```

* (Note: or use vite as per the previous section)

## Step 2: Install Redux Toolkit and Dependencies

* Next, install Redux Toolkit and React-Redux:

```bash

npm install @reduxjs/toolkit react-redux @types/react-redux

```

## Step 3: Define Redux Slices with TypeScript

* Create a TypeScript file for your Redux slice. For example, let's create a counter slice:

\*\*src/features/counterSlice.ts\*\*

```typescript

import { createSlice, PayloadAction } from '@reduxjs/toolkit';

// Define a type for the slice state

interface CounterState {

value: number;

}

// Initial state

const initialState: CounterState = {

value: 0,

};

// Create a slice

const counterSlice = createSlice({

name: 'counter',

initialState,

reducers: {

increment(state) {

state.value += 1;

},

decrement(state) {

state.value -= 1;

},

// Action with payload

incrementByAmount(state, action: PayloadAction<number>) {

state.value += action.payload;

},

},

});

// Export actions and reducer

export const { increment, decrement, incrementByAmount } = counterSlice.actions;

export default counterSlice.reducer;

```

## Step 4: Create Redux Store with TypeScript

\*\*src/app/store.ts\*\*

```typescript

import { configureStore } from '@reduxjs/toolkit';

import counterReducer from '../features/counterSlice';

const store = configureStore({

reducer: {

counter: counterReducer,

// other reducers can be added here

},

});

export type RootState = ReturnType<typeof store.getState>;

export type AppDispatch = typeof store.dispatch;

export default store;

```

## Step 5: Set Up Redux Provider in Your App

\*\*src/index.tsx\*\*

```typescript

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import App from './App';

import store from './app/store';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

```

## Step 6: Use Redux State and Actions in TypeScript Components

Now, let's use Redux state and actions in your React components.

\*\*src/App.tsx\*\*

```typescript

import React from 'react';

import { useSelector, useDispatch } from 'react-redux';

import { RootState, AppDispatch } from './app/store';

import { increment, decrement, incrementByAmount } from './features/counterSlice';

function App() {

const dispatch: AppDispatch = useDispatch();

const count = useSelector((state: RootState) => state.counter.value);

return (

<div>

<div>

<button onClick={() => dispatch(increment())}>Increment</button>

<span>{count}</span>

<button onClick={() => dispatch(decrement())}>Decrement</button>

</div>

<div>

<button onClick={() => dispatch(incrementByAmount(5))}>Increment by 5</button>

</div>

</div>

);

}

export default App;

```

## Step 7: Run Your Application

Finally, run your application:

```bash

npm start

```

Now you have a React application with Redux Toolkit integrated with TypeScript! TypeScript will help you catch type-related errors and provide better code hints and auto-completion, improving the development experience. Adjust the examples above according to your specific application needs and additional Redux slices.

# Typescript + React + Redux Toolkit + Redux Toolkit Query

## ChatGPT explanation

* Certainly! If you're integrating TypeScript into a React application that uses Redux Toolkit Query (RTK Query) along with Redux Toolkit, you'll need to set up TypeScript for both Redux Toolkit (for managing global state) and RTK Query (for data fetching and caching). RTK Query works seamlessly with TypeScript, and its typings are well-supported.
* Here's an example of how you would integrate TypeScript into a React application using Redux Toolkit for state management and RTK Query for data fetching:

## Step 1: Setting Up a TypeScript React Application

* Create a new React application with TypeScript:

```bash

npx create-react-app my-app --template typescript

cd my-app

```

## Step 2: Install Dependencies

* Install Redux Toolkit, RTK Query, React-Redux, and TypeScript typings for React-Redux:

```bash

npm install @reduxjs/toolkit @reduxjs/toolkit/query react-redux @types/react-redux

```

## Step 3: Define Redux Slices with TypeScript

* Create a TypeScript file for your Redux slice:

\*\*src/features/counterSlice.ts\*\*

```typescript

import { createSlice, PayloadAction } from '@reduxjs/toolkit';

interface CounterState {

value: number;

}

const initialState: CounterState = {

value: 0,

};

const counterSlice = createSlice({

name: 'counter',

initialState,

reducers: {

increment(state) {

state.value += 1;

},

decrement(state) {

state.value -= 1;

},

incrementByAmount(state, action: PayloadAction<number>) {

state.value += action.payload;

},

},

});

export const { increment, decrement, incrementByAmount } = counterSlice.actions;

export default counterSlice.reducer;

```

## Step 4: Create Redux Store with TypeScript

\*\*src/app/store.ts\*\*

```typescript

import { configureStore } from '@reduxjs/toolkit';

import { setupListeners } from '@reduxjs/toolkit/query/react';

import { counterSliceReducer } from '../features/counterSlice';

import { api } from './services/api';

const store = configureStore({

reducer: {

counter: counterSliceReducer,

[api.reducerPath]: api.reducer,

},

middleware: (getDefaultMiddleware) =>

getDefaultMiddleware().concat(api.middleware),

});

setupListeners(store.dispatch);

export type RootState = ReturnType<typeof store.getState>;

export type AppDispatch = typeof store.dispatch;

export default store;

```

## Step 5: Define RTK Query API Service with TypeScript

* Create a service file for your RTK Query API:

\*\*src/app/services/api.ts\*\*

```typescript

import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';

interface Post {

userId: number;

id: number;

title: string;

body: string;

}

export const api = createApi({

reducerPath: 'api',

baseQuery: fetchBaseQuery({ baseUrl: 'https://jsonplaceholder.typicode.com/' }),

endpoints: (builder) => ({

getPosts: builder.query<Post[], void>({

query: () => 'posts',

}),

}),

});

export const { useGetPostsQuery } = api;

```

## Step 6: Set Up Redux Provider in Your App

\*\*src/index.tsx\*\*

```typescript

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import App from './App';

import store from './app/store';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

```

## Step 7: Use Redux State, Actions, and RTK Query in TypeScript Components

* Now you can use Redux state, actions, and RTK Query hooks in your TypeScript components:

\*\*src/App.tsx\*\*

```typescript

import React from 'react';

import { useSelector, useDispatch } from 'react-redux';

import { RootState, AppDispatch } from './app/store';

import { increment, decrement, incrementByAmount } from './features/counterSlice';

import { useGetPostsQuery } from './app/services/api';

function App() {

const dispatch: AppDispatch = useDispatch();

const count = useSelector((state: RootState) => state.counter.value);

const { data: posts, error, isLoading } = useGetPostsQuery();

return (

<div>

<div>

<button onClick={() => dispatch(increment())}>Increment</button>

<span>{count}</span>

<button onClick={() => dispatch(decrement())}>Decrement</button>

</div>

<div>

<button onClick={() => dispatch(incrementByAmount(5))}>Increment by 5</button>

</div>

<div>

{isLoading ? (

<p>Loading...</p>

) : error ? (

<p>Error: {error.message}</p>

) : (

<ul>

{posts?.map((post) => (

<li key={post.id}>{post.title}</li>

))}

</ul>

)}

</div>

</div>

);

}

export default App;

```

* In this example:
  + `useSelector` is used to select the counter value from the Redux store (`RootState` is used to define the type of the Redux store state).
  + `useDispatch` is used to get the dispatch function with the correct type (`AppDispatch`).
  + `useGetPostsQuery` is an RTK Query hook generated by `createApi`, which fetches and manages the data from the API endpoint (`data`, `isLoading`, and `error` are inferred types).
* Ensure that you import and use the correct types (`RootState`, `AppDispatch`) from the Redux store in your components.
* This setup allows you to leverage the benefits of TypeScript for type safety and better code completion when working with Redux Toolkit and RTK Query in your React application. Adjust the examples according to your specific API endpoints and application requirements.