

Part3 - Best Practices in JavaScript & Typescript

Good parts and what to avoid, new features, performance tips etc.
Collective knowledge - easy to read, solid understanding how to use TS

Clean Architecture: Patterns, Practices, and Principles

Azat Satklycov

GitHub Repo: <https://github.com/azatsatklichov/nodejs-app> (See all in-deep examples – JS, TS and Node.JS related)

Object Mutability

Javascript primitives are immutable. BUT, Javascript Objects are mutable even if defined as constant. You can update its values or add new elements with new key/value pairs.

```
const myObj = {  
  "key": "value",  
  "someKey": 5,  
  "anotherKey" : true  
}  
// Let's update one of the keys on myObject  
myObj["key"] = "NEW VALUE"
```

Non-mutable (Immutable) objects

The only time an object's mutability changes is if you change it yourself. All objects have three hidden properties that configure (**Writable**, **Enumerable**, **Configurable**) their mutability along with their value, by default, all are true.

Some useful utility methods exist to change these properties – works on **single nesting level**

Object.freeze() - change an object to read-only, becomes { **writable**: false, **configurable**: false } **//props can't be modified**
Another similar property, **Object.seal()**, will instead set an object to { **configurable**: false } **//can't delete or add new prop.**

```
> let myObject = { "name" : "John" }  
Object.freeze(myObject)  
myObject["age"] = 5  
myObject["name"] = "Johnny"  
console.log(myObject)  
  
▶ {name: 'John'}
```

```
> const myObject = {}  
  
Object.defineProperty(myObject, "name", {  
  value: "John",  
  writable: false,  
})  
  
< ▶ {name: 'John'}
```

Use readonly types (``readonly`` and ``ReadonlyArray``) to Avoid Errors Associated with Mutation

```
const immutable = 'cannot be changed';
```

```
let mutable = 'can be changed';
```

```
const mutable: number[] = [1, 2, 3];
```

```
mutable.push(4);
```

```
mutable[2] = 5;
```

```
const immutable: readonly number[] =  
  [1, 2, 3];
```

```
const immutable: ReadonlyArray<number> =  
  [1, 2, 3];
```

◀ `const` variables are inherently immutable

◀ `let` variables can be changed at any time

Class properties/objects/arrays, etc are also **mutable by default**

◀ Arrays are inherently mutable, even declared with `const`

◀ We can mutate with array methods like `push`

◀ Or we can use square bracket notation with assignment

◀ We can use the ``readonly`` modifier here too

◀ Alternative utility type, use ``readonly`` for readability and succinctness

There are two important caveats to know about with the **readonly** property modifier and **Readonly<T>**

First: They are shallow - .e.g single nesting level











```
interface Outer {
  inner: {
    x: number;
  }
}
const obj: Readonly<Outer> = { inner: { x: 0 } };
obj.inner = { x: 1 };
// ~~~~ Cannot assign to 'inner' because it is a
// read-only property
obj.inner.x = 1; // OK
console.log(obj)
```

```
//create a type alias, inspect it
type T = Readonly<Outer>;
// ^? type T = {
//   readonly inner: {
//     x: number;
//   };
// }
//readonly modifier on inner but not on x
```

There is no built-in support for deep readonly types, but it is possible to write a generic type to produce them. Better use [ts-essentials](#) (or alternative) library type e.g. **DeepReadonly**

Second: Readonly affects only properties - If you apply it to a type with methods that mutate (e.g. pop, shift) the underlying object, it won't remove them

```
const date: Readonly<Date> = new Date();
date.setFullYear(2037); // OK, but mutates date!
```

	ts-essentials	
	type-fest	
	typelevel-ts	
	typical-ts	
	utility-types	

If you want both a mutable and immutable version of a class, you'll generally need to separate them yourself.

e.g. Array and ReadonlyArray

```
>tsc .\3-readonly.ts | node .\3-readonly.js
```

```
const names: readonly string[] = ["Oraz"];
names.push("Jack"); //readonly prevents arrays from being changed
// Error: Property 'push' does not exist on type 'readonly string[]'.
```

Use Utility Types Where Applicable!!!

TS comes with a large number of types that can help with some common type manipulation, usually referred to as utility types.

Partial

Partial changes all the properties in an object **to be optional**.

We can use the Partial utility to avoid creating a whole new type

```
interface Point {  
  x: number;  
  y: number;  
}  
// `Partial` allows x and y to be optional  
let pointPart: Partial<Point> = {};  
pointPart.x = 10;
```

Required

Required changes all the properties in an object **to be required**

The Required utility takes a type and returns a new type where all properties are mandatory

```
interface Car {  
  make: string;  
  model: string;  
  mileage?: number;  
}  
  
let myCar: Required<Car> = {  
  make: 'Ford',  
  model: 'Focus',  
  mileage: 12000 // `Required` forces mileage to be defined  
};
```

Record

Record is a shortcut to defining an object type with a **specific key type and value** type.

The Record utility describes an object-like structure with strings as keys and any values

```
const nameAgeMap: Record<string, number> = {  
  'Alice': 21,  
  'Bob': 25  
};
```

Record<string, number> is equivalent to { [key: string]: number }, Key type is string;

Record<key, result> → {[key:string]:result}

Pick

Pick removes all but the specified keys from an object type.

Omit

Omit inverse of **Pick**

```
type NewType = Omit<OriginalType, 'Key1' | 'Key2'>
```

```
interface Person {
  name: string;
  age: number;
  location?: string;
}
const bob: Pick<Person, 'name'> = {
  name: 'Bob'
};
```

```
interface Person { name: string; age: number;
gender: string; address: string; }
```

```
type NewPerson = Omit<Person, 'address'>;
```

```
const person: NewPerson = {
name: 'John', age: 30, gender:
'male' };
```

Exclude

Exclude removes types from a union.
Not confuse with Omit. This works with
Union Types

```
type Primitive = string | number | boolean
const value: Exclude<Primitive, string> = true; // a string type cannot be used anymore
```

ReturnType

ReturnType extracts the return type of a function type.

```
type PointGenerator = () => { x: number; y: number; };
const point: ReturnType<PointGenerator> = {
  x: 10,
  y: 20
};
```

Parameters

Parameters extracts the parameter types of a function type as an array.

```
type PointPrinter = (p: { x: number; y: number; }) => void;
const point: Parameters<PointPrinter>[0] = {
  x: 10,
  y: 20
};
```

Readonly

Readonly is used to create a new type where all properties are readonly, meaning they cannot be modified once assigned a value.
Keep in mind TypeScript will prevent this at compile time, but in theory since it is compiled down to JavaScript you **can still override a readonly property**

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

```
const person: Readonly<Person> = {
  name: "Dylan",
  age: 35,
};
person.name = 'polo'; // prog.ts(11,8): error TS2540: Cannot assign to 'name' because it is a read-only property.
```

Indexed Access Types

We can use an ***indexed access type*** to look up a specific property on another type:

```
type Adam = { age: number; name: string; alive: boolean };
type Age = Adam["age"]; //type Age = number

//The indexing type is itself a type, so we can use unions, keyof, or other types entirely:
type I1 = Adam["age" | "name"]; //type I1 = string | number
type I2 = Adam[keyof Adam]; //type I2 = string | number | boolean

type AliveOrName = "alive" | "name";
type I3 = Adam[AliveOrName]; //type I3 = string | boolean

//You'll even see an error if you try to index a property that doesn't exist:
type I4 = Adam["alma"]; //Property 'alma' does not exist on type 'Adam'
```

Indexing with an arbitrary type is using
number to get the type of an array's elements

```
type PersonA = typeof MyArray[number];
```

You can only use types when indexing, meaning you can't use a const to make a variable reference:

```
const key = "age";
//type AgeW = Adam["age"]; //OK
type AgeW = Adam[key]; //Type 'key' cannot be used as an index type
```

You can't use indexed access type as a value

```
console.log(Adam["age"]) //'Adam' only refers to a type, but is being used as a value
```

Mapped Types

When you don't [want to repeat yourself](#), sometimes a type needs to be based on another type. **Mapped types** build on the syntax for **index signatures**, which are used to declare the types of properties which have not been declared ahead of time:

A mapped type is a generic type which uses a union of PropertyKeys (frequently created [via a keyof](#)) to iterate through keys to create a type:

```
type OptionsFlags<Type> = {
  [Property in keyof Type]: boolean;
};

type Features = {
  darkMode: () => void;
  newUserProfile: () => void;
};

type FeatureOptions = OptionsFlags<Features>;
type ActionOptions = OptionsFlags<Actions>;
//just mouse hover over
```

```
type FeatureOptions = {
  darkMode: boolean;
  newUserProfile: boolean;
}

type FeatureOptions = OptionsFlags<Features>;
```

Mapping Modifiers

There are two additional modifiers which can be applied during mapping: **readonly** and **? (optional)** which affect mutability and optionality respectively. You **can remove or add** these modifiers by prefixing with **-** or **+**. If you don't add a prefix, then **+** is assumed.

```
type CreateMutable<Type> = {
  -readonly [Property in keyof Type] : Type[Property];
};

type LockedAccount = {
  readonly id: string;
  readonly name: string;
};

type UnlockedAccount = CreateMutable<LockedAccount>;
```

```
type Concrete<Type> = {
  [Property in keyof Type]-?: Type[Property];
};

type MaybeUser = {
  id: string;
  name?: string;
  age?: number;
};

type User = Concrete<MaybeUser>;
```


e.g. Passing arguments with wrong order can be costly

```
const comments =  
getCommentsForPost(user.id, post.id)  
// This is OK for Typescript
```

```
async function getCommentsForPost(postId: string, authorId:  
string) {  
    const response =  
    api.get(`/author/${authorId}/posts/${postId}/comments`)  
    return response.data  
}
```

Branded Types

Provide a way to create deeper specificity and [uniqueness for modeling](#) your data.
Use cases: Custom validation, Domain Modeling, API req/res,

Brands can be added to a type using a union of the base type and an object literal with a branded property.

```
type Brand<K, T> = K & { __brand: T }  
type UserID = Brand<string, "UserId">  
type PostID = Brand<string, "PostId">  
type CommentID = Brand<string, "CommentId">
```

```
async function  
getCommentsForPost2(postId: PostID, authorId: UserID) {  
    const response = await  
    api.get(`/author/${authorId}/posts/${postId}/comments`)  
    return response.data }  
}
```

```
const comments2 = getCommentsForPost2(user.id, post.id) // // ✗ This fails (see code)
```

Still not be present on runtime, so possible to duplicate branded types etc. Use stronger impl.

```
declare const __brand: unique symbol  
type Brand3<B> = { [__brand]: B }  
export type Branded<T, B> = T & Brand3<B>  
  
type UserID3 = Branded<string, "UserId">  
type PostID3 = Branded<string, "PostId">  
type CommentID3 = Branded<string, "CommentId">
```

```
const comments3 =  
getCommentsForPost3(user3.id, post3.id)  
// ✗ This fails
```

Classes, Access modifiers, Parameters, Methods

ECMAScript 2015, also known as ES6, introduced JS Classes.
In JS, you can create Objects without classes, **not like in JAVA**

The class Statement, class Expression

```
class MyClass {} //class statement, e.g. new MyClass()
```

```
let myClass = class {};//class expression, e.g. new myClass()
```

Access modifiers (on fields, methods), Encapsulation.

Public, [Also **public** if not def.], **#Private** (in JS only #, remains at Runtime), Protected

Parameters can be: mandatory, default, optional

```
class Point {  
  x: number;  
  y: number;  
  constructor(x?: number, y?: number) {  
    this.x = x;  
    this.y = y;  
  }  
}
```

Constructor pattern (shorthand way). No multiple constructor (use optional params, or static factory methods)

TS compiler will generate fields implicitly with same name & initialize.

```
class Point {  
  constructor(private x?: number, private y?: number) { //can be also public, then mutable,..  
  }  
}
```

Methods

By default, all class methods are public. That means they can be **changed/redefined/deleted** outside of the class. You don't want a class editable like this.

```
Utility.myMethodNotReturnText = function() {  
  return "classDetails - corrupted"  
}
```

You can define methods or fields as **private or static**
Static fields, which cannot be accessed on a new instance of a class itself but only on the original class (which we'll look at in more detail soon) .

Not like in JAVA



```
let callUtility = new Utility  
console.log(callUtility.classDetails())
```

r-class.js , r-class2.js

Getters/Setters, Properties

How to access private fields?

1) Getters & Setters - Getters and setters are just syntactic sugar – they make your code a little easier to understand from the outside

3a-point-class.ts, 3a-props-class.ts, js-getters-setters.js

Concept of Properties: camelCase fields (get X clashes with x, so use _x)

At the top level of the class body, variables can be defined within a class without variable keywords. That's because they **act like the properties** of an object.

```
//concept of properties
let p5 = new Point5(23, 40);
//p5.x;
x = p5.x; //like properties
console.log(X)
console.log(Y)
console.log(plot)
```

Other Usage of '_' : E.g. app.get('/forms', (_req, res) => {..}) to defer warning “req” is declared but its never read (next page).

Inheritance

Class Inheritance via Extends: Since classes are mostly syntactic sugar on top of objects, they also have typical prototypical inheritance too. If a class extends another class, the class which extends will simply become the prototype of the child class.

Also not that TS inheritance supports both field and method overriding level whereas JAVA only via method overriding.

```
//see examples
console.log(v1.info())
console.log(v2.info())
```

tsc .\3a-props-class.ts | node .\3a-props-class.js
3b-class-inheritance.ts, r-class3-inheritance.js

Declaring Parameters (default, optional, rest, no-warning).

In **JS** by default all parameters are OPTIONAL. But in Typescript all parameters are **required** by default.

```
//default param
var myFun1 = function (y: string = "I am") {
    return y;
};

console.log(myFun1("23")); //23
console.log(myFun1()); //I am
```

```
//optional param
var myFun11 = function (y?: string) {
    return y; //y?: number | undefined
};

console.log(myFun11("23")); //23
console.log(myFun11()); //undefined
```

```
//rest parameters
var myFun3 = function (x: number, ... ids: number[]) { //var-args in JAVA
    //tbd
}







myFun3(2);
myFun3(2, 3);
myFun3(2, 55, 453);
```

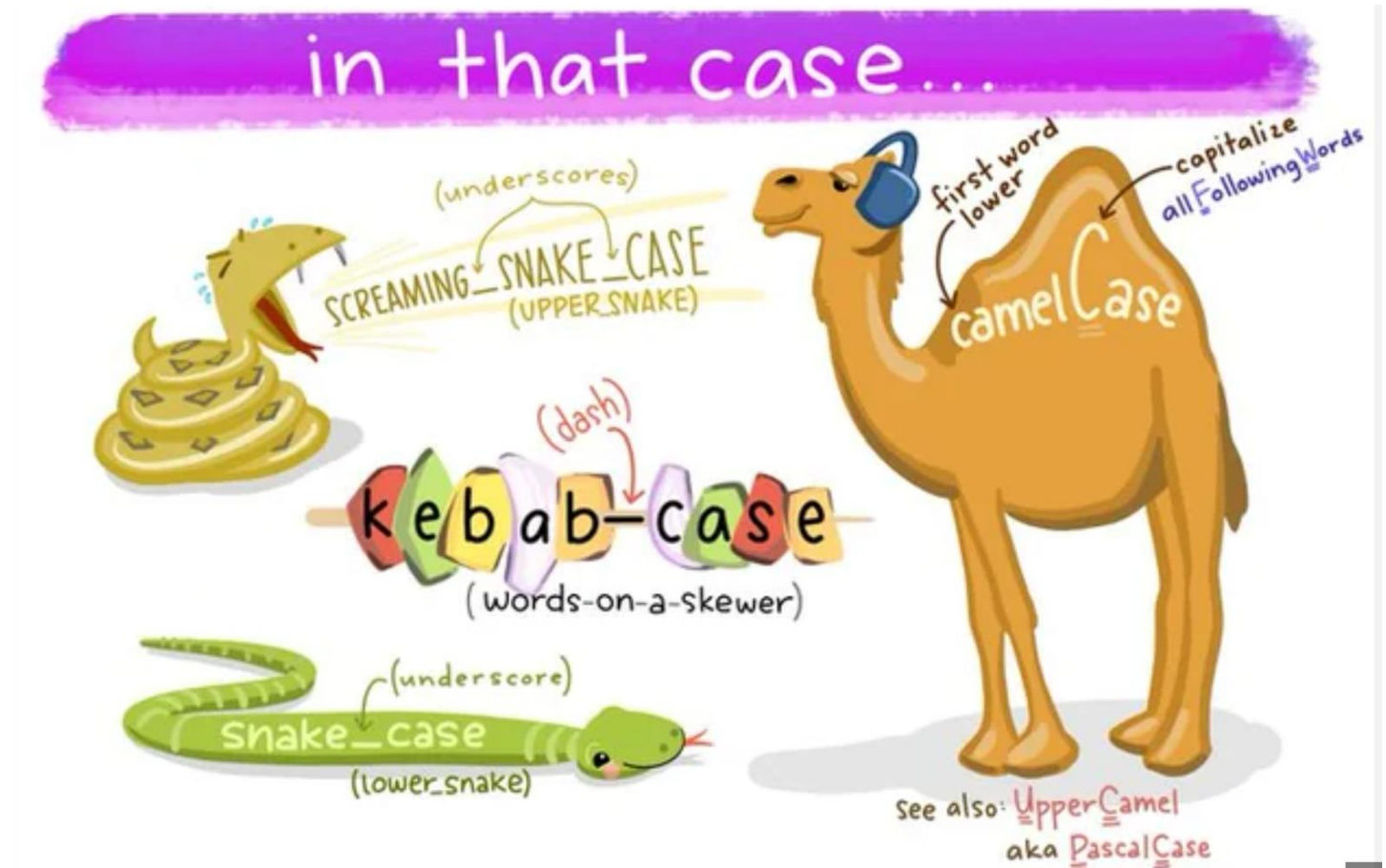
```
//Underscored Variable. No warning shown for un-used variables, e.g. req, res objects, ...
function vars(_variable: string) {
    console.log('Just see, no warning even variable not used' );
}
```

Variable and Function Naming Convention

Camel Case, Pascal Case, Snake Case - underscoring, and Kebab Case, ...

When naming variables, functions, end-points (rest resources) any of the naming paradigms are fine, BUT it is important to be consistent everywhere

	myGreatVariableName Camel Case
	my/great/variable/name Slash Case
	my_great_variable_name Snake Case
	MyGreatVariableName Pascal Case
	My_Great_Variable_Name Cobra Case
.	my.great.variable.name Dot Case
-	my-great-variable-name Dash Case
	my great variable name Separate Words



The Polymorphic `this`

```
export class Calculator {
  private result: number = 0;

  public add(num: number): this {
    this.result += num;
    return this;
  }

  public subtract(num: number): this {
    this.result -= num;
    return this;
  }

  public multiply(num: number): this {
    this.result *= num;
    return this;
  }

  public reportResult(): number {
    return this.result;
  }
}
```

```
const simpleCalc = new Calculator();

simpleCalc
  .add(5)
  .subtract(1)
  .multiply(2)
  .reportResult();
```

◀ Consider a simple Calculator class

◀ At the end of the method we return `this`

◀ Each method does a simple calculation and returns `this`

◀ **Not all methods need to be chainable** in a fluent API

◀ It's very easy to use the calculator

◀ We can **chain multiple methods**

◀ This method ends the chain

The `this` object can refer to different things in different contexts
We don't need to do anything special to make `this` polymorphic

Know the Differences Between type and interface

type or **interface** are in type space, while those introduced in a **const** or **let** declaration are values.

type `T1 = 'string literal'`; **const** `v1 = 'string literal'`; class and enum constructs introduce both a type and a value.

If you want to define a named type in TypeScript, you have two options: **Type Aliases** or **Interfaces** (or **Class**)

```
type TState = {  
  name: string;  
  capital: string;  
};
```

```
interface IState {  
  name: string;  
  capital: string;  
}
```

Which should you use, type or interface?

You should know the distinctions and how to write the same types using both, so that you'll be comfortable reading both. For new code where you need to pick a style/design, the general rule of thumb is to use interface where possible, using type either where it's required (e.g., **union types**) or has a cleaner syntax (e.g., function types).

Similarities - Two State types (Types alias and Interface are nearly indistinguishable.)

The **errors you get** from excess property checking are same

```
const s1: TState = {  
  name: 'Wyoming',  
  capital: 'Cheyenne',  
  population: 578_000  
  // Object literal may only specify known properties,  
  // and 'population' does not exist in type 'TState'  
};
```

```
const s2: IState = {  
  name: 'Wyoming',  
  capital: 'Cheyenne',  
  population: 578_000  
  // Object literal may only specify known properties,  
  // and 'population' does not exist in type 'IState'  
};
```

You can use an **index signature** (can be used for **records**)
with both interface and type.
Isn't his a RECORD here??

```
type TDict = { [key: string]: string };  
interface IDict {  
  [key: string]: string;  
}
```

You can also **define function types**

```
type TFn = (x: number) => string;  
interface IFn {  
  (x: number): string;  
}  
const toStrT: TFn = x => '' + x; toStrT(343); // OK  
const toStrI: IFn = x => '' + x; toStrI(341); // OK
```


Both types can be generic

```
type TBox<T> = {  
  value: T;  
};
```

```
interface IBox<T> {  
  value: T;  
}
```

An **interface** can extend a **type** and a **type** can extend an **interface**

```
interface IStateWithPop extends TState {  
  population: number;  
}
```

```
type TStateWithPop = IState & { population: number; };
```

A class can implement either an interface or a simple type

```
class StateT implements TState {  
  name: string = '';  
  capital: string = '';  
}
```

```
class StateI implements IState {  
  name: string = '';  
  capital: string = '';  
}
```

Differences - Two State types (Types alias and Interface are nearly indistinguishable).

//1. there **are union types** but **no union interfaces**

```
type AorB = 'a' | 'b'; //this still can be simulated, see code
```

//This type cannot be expressed with interface.

```
type NamedVariable = (Input | Output) & { name: string };
```

A **type** is, in general, **more capable than an interface**.
It can be a union, and it can also take advantage of fancy type-level features like mapped types, conditional types

//2. **interface and extends** give a bit **more error checking** than **type** and **&**:

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

```
type TPerson = Person & { age: number; }; // no error, unusable type
```

```
interface IPerson extends Person {
```

```
  // ~~~~~ Interface 'IPerson' incorrectly extends interface 'Person'.
```

```
  // Types of property 'age' are incompatible.
```

```
  // Type 'number' is not assignable to type 'string'.
```

```
  age: number;
```

```
}
```

Generally you want more **safety checks**, so this is a good reason to use extends with interfaces.

//3. Type aliases are the natural way to express tuple and array types:

```
type Pair = [a: number, b: number];
```

```
type StringList = string[];
```

```
type NamedNums = [string, ...number[]];
```

4-named-types-differences.ts

//4. Declaration Merging

Not in JAVA

An **interface** does have some abilities that a **type** doesn't, however. One of these is that an interface can be *augmented*.
a.k.a Declaration merging – compiler merges two separate declarations declared with same name into one

```
interface Mashyn {
  name: string;
  go(): () => void;
}
//somewhere in app another Mashyn interface
interface Mashyn {
  color: string;
  stop(): () => void;
}
//TS compiler merges it and sees as single interface. Code completion support, ..
class Volga implements Mashyn {
  name: string;
  go(): () => void {
    throw new Error("Method not implemented.");
  }
  color: string;
  stop(): () => void {
    throw new Error("Method not implemented.");
  }
}
```

TS itself uses declaration merging to model the different versions of JS's standard library. The Array interface, e.g. ES5 is defined in lib.es5.d.ts. But if you target ES2015, TypeScript will also include lib.es2015.core.d.ts e.g. 4 new methods.

Can be merged: Interfaces, Enums, Namespaces, Namespaces with classes | functions | enums

Can not be merged: Classes with classes. Workaround is [Mixin concept](#) (like traits in Scala)

4-decl-merging.ts

Avoid Including null or undefined in Type Aliases

Is optional chain (?.) necessary? Could user ever be null?

```
function getCommentsForUser(comments: readonly Comment[], user: User) {  
    return comments.filter(comment => comment.userId === user?.id);  
}
```

If it's a type alias that allows null or undefined, then the **optional chain is needed**.

```
type User = { id: string; name: string; } | null;
```

If it's a simple object type (see interface), **then it's not**.

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

General rule - better to avoid type aliases that allow null or undefined values

If you must include null in a type alias for some reason, do readers of your code a favor and use a name that's unambiguous:

```
type NullableUser = { id: string; name: string; } | null;
```

OR, better use Interfaces (as nuanced type)

```
interface NullableStudent {  
    name: string;  
    ageYears: number | null;  
}  
  
interface Student extends NullableStudent {  
    ageYears: number;  
}
```

```
const s1:NullableStudent = {name:"Ole", ageYears:null}  
const s2:Student = {name:"Ole", ageYears:null}  
//Type 'null' is not assignable to type 'number'.ts(2322)
```

4-type-aliases-avoid-null-undefined

Type Definition Files, Ambient Declaration

- Once you work with Javascript & DOM (Table, Input .. elements)
- **lib.d.ts** is referenced by default for the **DOM** and JavaScript in TS.
- Ambient Declarations (**declare**) do not appear **anywhere in the JavaScript**
- [*.d.ts files](#) not to run but to give context for **code-hints, err-detection**
- Primarily used as a TS wrapper for JS libraries

TS

```
declare var document;  
  
document.title = "Hello";
```

Javascript

```
document.title = "Hello";
```

Triple-slash directives are single-line comments containing a single XML tag. The contents of the comment are used as **compiler directives**.

```
/// <reference path="..." />  
/  
/// <reference types="..." />  
/  
/// <reference lib="..." />
```

E.g. `var table: HTMLTableElement = document.createElement('table');`

5-ambient.ts
5-ambient2.ts

- Once working with third-party libs (jQuery, lodash etc.) you need *.d.ts file. E.g.

```
///</// <reference path="lodash.d.ts" />
```

//See more on [Triple-Slash](#) Directives

```
import * as _ from "lodash";
```

```
console.log(_.snakeCase('UEFA Champions League')); //uefa_champions_league
```

Type Definition Files. Typings

- Get *.d.ts sources from GitHub (even you can contribute via PR ;)

<https://github.com/borisyankov/DefinitelyTyped>, <http://definitelytyped.org/>

- Tools to manage: direct download from GitHub, Nuget, **tsd**, **typings**
- 1-way: `npm i --save lodash --save-dev @types/lodash`
 - > npm install --save-dev @types/jquery
- 2-way: using **tsd.json** [[deprecated](#)] – Find&download *.d.ts files, keeps all /// in single file
`npm install tsd -g`, then `tsd install lodash --save`, or `tsd install jquery --save` // typings folder
- 3-way: **typings.json** is new and like tsd but gets files from multiple sources [GitHub](#), **SVN**,
... add typings folder. Configure tsconfig.json files:["typings/main.d.ts"] to remove ///<...
Also declared: Deprecation Notice: Regarding TypeScript@2.0 , use 1-way: `npm install @types/<package>`

- `npm install typings --global`
- `typings install jquery --save`

> typings -v

2.1.1

Functions in TypeScript Versus JavaScript

TypeScript

Types (of course!)

Arrow functions

Function types

Required and optional parameters

Default parameters

Rest parameters

Overloaded functions

JavaScript

No types

Arrow functions (ES2015)

No function types

All parameters are optional

Default parameters (ES2015)

Rest parameters (ES2015)

No overloaded functions

Function Types

//function expression

```
let IdGenFunc: (chars: string, nums: number) => string;  
IdGenFunc = (name: string, id: number) => { return id + name; }
```

```
let myID: string = IdGenFunc('gulssirin', 63);// 63gulssirin
```

//classic function

```
function PubMsg(year: number): string {  
    return "Pub: " + year;  
}
```

//lambdas – fat arrow

```
let pubFunc: (someYear: number) => string;
```

```
pubFunc = PubMsg;
```

```
let msg: string = pubFunc(2022);
```


Overloaded Functions

In TS once types are removed during transpilation to JS, this adds ambiguity ..

```
//you can not just DO overload functions as in JAVA
function getId(user: string) {}
function getId(user: string) {}

//define
// 'user' is declared but its value is never read. ts(6133)
// Duplicate function implementation. ts(2393)
```

Defining overloaded functions – NOT possible as like in JAVA. But you can do:

//1. dummy way, just use rest-parameters, not fulfills the overloading need

//2. You can provide multiple type signatures for a function, with only a single implementation:

```
function getId(user: string): string[];
function getId(active: boolean): string[];
```

```
function getId(factor: any): string[] {
    if (typeof factor == 'string') { //narrowing the scope, context aware, Java 14 came this in Java 14-17
        //tbd
    } else if (typeof factor == 'boolean') {
        //tbd
    }
    return []; //TBD result
};
```

Class Expressions, Mixins

Common practice: In JS function expressions were used a lot.
What about CLASS expression in Typescript?

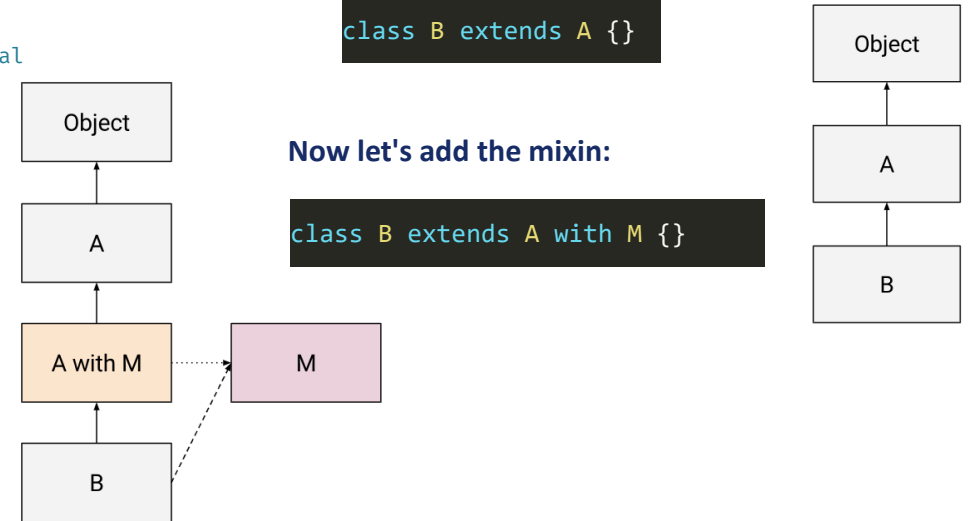
```
abstract class Animal {  
    abstract swim(txt: string): void;  
}  
//giving a class name is optional  
let Dolphin = class extends Animal { //can be Dolphin class extends Animal  
    swim(txt: string): void {  
        console.log('swim like ' + txt);  
    }  
}  
let myDolphin = new Dolphin(); //can be new dolphin  
myDolphin.swim('Dolphin');
```

Typescript's best [mixin](#) support is done via the [class expression pattern](#).

```
class B extends A {}
```

Now let's add the mixin:

```
class B extends A with M {}
```



```
//how to use class expression in extension  
class Sharq extends class {name : string} { //not confuse with object, this is declaration  
    elasmobranchii : string;  
}  
let mySharq = new Sharq();  
mySharq.elasmobranchii = 'elasmobranchii Sharq';  
mySharq.name = 'Alpha';
```

class with **no name**, like **anonymous class in Java**, but not exact one

5-class-expressions.ts

Destructing assignments

```
let apples: string[] = ['Granny Smith', 'Opal', 'Goldspur', 'Ligol', 'Melba'];
```

```
let apple1 = apples[0];
let apple2 = apples[1];
let apple3 = apples[3];
console.log(apple1); //Granny Smith
```

```
//destructing arrays
let [jablko1, jablko2, jablko3] = apples;
console.log(jablko1); //Granny Smith
```

```
// const PI = Math.PI;
// const E = Math.E;
// const SQRT2 = Math.SQRT2;
```

```
const { PI, E, SQRT2 } = Math;
```

```
const myObj = {
  z: undefined,
  y: 4,
  x: 3
}
```

```
const { x, y, z=5 } = myObj
```

```
console.log(y) // 4
```

```
console.log(z) // 5, when destructing, default value used
```

Destructuring assignment allows you to extract properties from objects and elements from arrays into distinct variables easily. Useful to only access to needed variables.

```
let car = {
  model: 'Skoda Fabia Kombi',
  karoserie: 'Kombi',
  assembly: 'Mlada Boleslav'
};
// let model = car.model;
// let karoserie = car.karoserie
// let assembly = car.assembly;
```

```
//property names exact match. UNLESS destructing ARRAY
let {model, karoserie, assembly} = car;
console.log(model); //new variable
//if object props not match or different naming
let {model:znacka, karoserie:style, assembly:factory} = car;
console.log(znacka); //new variable
const [a, b] = [23, 63]; console.log(a); //23
```

Array and Object Destructuring in Function Parameters

Extracting values from arrays or objects passed as function parameters can be verbose. Use destructuring in function parameters to directly extract values.

```
const user = { name: 'Jane', age: 25 };
function greet({ name, age }) {
  return `Hello, ${name}! You are ${age} years old.`;
}
console.log(greet(user)); // "Hello, Jane! You are 25 years old."
```

Destructuring in function parameters allows you to directly extract values from objects or arrays passed to the function, making the code more concise and readable.

Default Values in Destructuring

Handling missing properties when destructuring objects can be cumbersome. Use default values in destructuring to provide fallback values.

```
const user = { name: 'Jane' };
const { name, age = 18 } = user;
console.log(name); // "Jane"
console.log(age); // 18
```

Parameter Destructuring in Callbacks

```
const users = [
  { id: 1, name: 'Jane' },
  { id: 2, name: 'John' },
];
users.forEach(({ id, name }) => {
  console.log(`User ID: ${id}, User Name: ${name}`);
});
```

Accessing properties of objects passed to callbacks can be verbose. Use destructuring in callback parameters for cleaner code.

Spread operator (Three Dots) – Can do Merging Objects and Arrays, Coercing Arrays into Objects, Passing Arrays as args to func.

```
let greenApples: string[] = ['Granny Smith', 'Lodi', 'Smeralda'];  
//using spread operator for merging  
let allApples: string[] = ['Opal', 'Goldspur', 'Ligol', 'Melba', ...greenApples];  
const [first, ...restOfItems] = [10, 20, 30, 40];  
const newArray = [...restOfItems];
```

5b-spread.ts
z-modern-js.js

Arguments can also be called in a function via array, using the three dots syntax

```
function words(word1, word2) {  
    return word1 + " " + word2  
}  
let validWords = [ "Hello", "John" ]  
console.log(words(...validWords)) // Hello John
```

```
const myObj = {  
    z: undefined,  
    y: 4,  
    x: 3  
}  
const { x, ...rest } = myObj  
console.log(rest)  
// Only shows z and y: { z: undefined, y: 4 }
```

Objects can also be merged in the same way – but if **duplicate keys** are found, the second object will overwrite the first

```
let user1 = { "name" : "John", age: 24 }  
let user2 = { "name" : "Joe" }  
let combineUsers = { ...user1, ...user2 }  
console.log(combineUsers) // { "name" : "Joe", age: 24 }
```

Using the spread syntax on an array inside of an object literal will turn it into an object, too – where the keys are the indices of the array. This provides a simple way to **convert an array to an object**:

```
let animals = [ "cats", "dogs" ]  
let objectAnimals = { ...animals }  
console.log(objectAnimals)  
// {0: 'cats', 1: 'dogs'}
```

Dynamic property, Template Strings

Accessing an object property with a dynamically-computed name

//DYNAMIC PROPERTIES

```
const dyno = "dynamo"; //can be provided at runtime
const LOG2E = Math.LOG2E;
const obje = {
  a: "oka",
  f1() {}, //function with object literal
  f2: () => {}, //arrow function
  [dyno]: 63, //not confuse with array ;), dynamo : 63
  LOG2E //shorter than LOG2E : LOG2E
};
console.log(obje.dynamo) //63
console.log(obje['dynamo']) //63
```

Using Functions for
Dynamic Property Keys

```
function getPropertyKey() {
  return 'quantity';
}

const product = {
  [getPropertyKey()]: 10,
};

console.log(product.quantity); // Outputs: 10
```

Templates-string (**interpolation, `\${dyno-exp}`**) (like multi-lines Java 13)

```
const cube = (a) => a * a * a;
const html = `
  <div>
    ${Math.random()}
    <br/>
    ${cube(3)};
  </div>
`;
```

a) Template literals make it easier to create strings with **concatenations**, embedded expressions and multi-line strings.

Other ways: b) 'dsd'+ 'sdsd' c) or use String **concat()** method

5b-dynamic-prop.ts

Template Literal Types (only in TS)

TLT build on [string literal types](#), and have the ability to expand into many strings via unions.

```
type EmailLocaleIDs = "welcome_email" | "email_heading";
type FooterLocaleIDs = "footer_title" | "footer_sendoff";
type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
//type AllLocaleIDz = "welcome_email_id" | "email_heading_id" | "footer_title_id" | "footer_sendoff_id"
```

For each interpolated position in the template literal, the unions are cross multiplied (hover over):

```
type AllLocaleIDs2 = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
type Lang = "en" | "ja" | "pt";
```

```
/**
 * For each interpolated position in the template literal, the unions are cross multiplied (hover over):
 */
type LocaleMessageIDs2 = "en_welcome_email_id" | "en_email_heading_id" | "en_footer_title_id" | "en_footer_sendoff_id" | "ja_welcome_email_id" | "ja_email_heading_id" | "ja_footer_title_id" | "ja_footer_sendoff_id" | "pt_welcome_email_id" | "pt_email_heading_id" | "pt_footer_title_id" | "pt_footer_sendoff_id";
type LocaleMessageIDs2 = `${Lang}_${AllLocaleIDs}`;
```

Inference with Template Literals - Get **benefit from all the information provided** in the original passed object.

```
type PropEventSource<Type> = {
  on<Key extends string & keyof Type>(
    eventName: `${Key}Changed`, callback: (newValue: Type[Key]) => void): void;
};
//watcher function
declare function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>;

const person = makeWatchedObject({ //create watched object with an ON-method
  firstName: "Saoirse",
  lastName: "Ronan",
  age: 26
});
```

```
//the callback will receive an argument of type string
person.on("firstNameChanged", newName => {
  console.log(`new name is ${newName.toUpperCase()}`);
});

//the callback will receive an argument of type number
person.on("ageChanged", newAge => {
  if (newAge < 0) {
    console.warn("warning! negative age");
  }
})

//Here we made on into a generic method.
```

5b-template-literal-types.ts

Type Guards, User defined Types

Compiler can do check more errors on narrowed block based on **type guards** [**typeof**, **instanceof**] .
Typeof is also used in Overloaded method implementation, ...

```
let x: string | number = 144;
if(typeof x === 'number') {
    //TYPE is NARROWED to NUMBER
    //(not exist in Java until Java 15)
} else {
    //narrowed to STRING. Compiler does this
}
```

Instanceof - works on other types, which has a constructor, ..

```
class Football {}
class Hockey {}
let sport: Football | Hockey = new Football();
if(sport instanceof Football){
    //narrowed to Football, so safe to use
}
```

Drawbacks:

Only used for specific types (string, number, boolean and symbol).

Instanceof - works on other types, which has a constructor

User defined Type Guards

```
interface Drink { taste: string}
//Java has sealed Classes concept, can be used for similar usecases
function isDrink(d : any) : d is Drink {
    return (<Drink> d).taste !==undefined;
}
let f = new Football();
if(isDrink(f)){
    console.log('Yes it is a drink type');
} else {
    console.log('It is not a drink');
}
```

Favor Type-guards over Type Assertions

Instead of using a type assertion to *tell* TypeScript what type a value is...

```
let a: string;
a = 1 as string;

// "Conversion of type 'number' to type
// 'string' may be a mistake because
// neither type sufficiently overlaps with
// the other."
```

- ◀ Add string type
- ◀ Use as operator to tell TypeScript to treat `1` as a string
- ◀ TypeScript prevents this kind of assertion



THANK YOU

References

https://www.w3schools.com/js/js_intro.asp

<https://medium.com/codex/good-to-know-parts-when-coding-with-javascript-996cd68563d3>

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

<https://medium.com/javascript-non-grata/the-top-10-things-wrong-with-javascript-58f440d6b3d8>

<https://blog.devgenius.io/45-javascript-super-hacks-every-developer-should-know-92aecfb33ee8>

<https://medium.com/@julienetienne/is-javascript-trash-part-1-5310ac4e20d0>

<https://github.com/danvk/effective-typescript>

<https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>