

Part2 - 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)

More pain-point examples in JS

```
//JavaScript's equality operator (==) coerces its operands, leading to unexpected behavior:
if (" " == 0) {
    // It is! But why??
    console.log('Trueeee')
}
if (1 < x < 3) {
    // True for *any* value of x!
    console.log('Trueeee');
}
```

```
//JavaScript also allows accessing properties which aren't present:
const obj = { width: 10, height: 15 };
// Why is this NaN? Spelling is hard!
const area = obj.width * obj.heighth;
```

If this code is in Typescript, static type checker complain:

Property 'heighth' does not exist on type '{ width: number; height: number; }'.

Did you mean 'height'?

```
console.log(4 / []); //Infinity
```

If this code is in Typescript, you get:

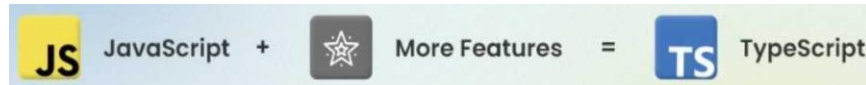
The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.

Template Literal Types (only in TS)

TypeScript	JavaScript
Types (of course!)	No types
Arrow functions	Arrow functions (ES2015)
Function types	No function types
Required and optional parameters	All parameters are optional
Default parameters	Default parameters (ES2015)
Rest parameters	Rest parameters (ES2015)
Overloaded functions	No overloaded functions

z-test.js , z-test.ts

Why Typescript



What brings TypeScript (identifies 15% of all JS errors), [tsc](#) (compiler), and [tsserver](#)(language server).
Typed Superset of JS in a syntactic sense, transpiled to ES5 | ES6 .. (compiles to vanilla Javascript - no type info).
But not all TS programs are valid JavaScript programs.

- ❑ Compile time check, Static type, Types Inference, Tuples, Generic, OO classes ES6 based (**see diagram**) etc.
- ❑ TS disallows some legal but questionable JS constructs such as calling functions with the wrong # arguments.
- ❑ TypeScript **never** changes the **runtime behavior** of JavaScript code.
- ❑ [TypeScript](#) 5.x [Updates](#) - actively maintained and updated by Microsoft.

➤ To get rid off Function **Spaghetti Code** (oldy JS code, e.g. calendar)
→ [Ravioli Code](#) ([JS Patterns](#)) (Each JS module has specific concern)

- Types declarations ([*d.ts](#)) - once using DOM, jQuery, lodash, ..
- How to [convert TS files to JS](#) ([tsc](#) vs. [Babel](#))

❖ TS alternatives – Dart, CoffeeScript, Applying JavaScript Patterns, ..

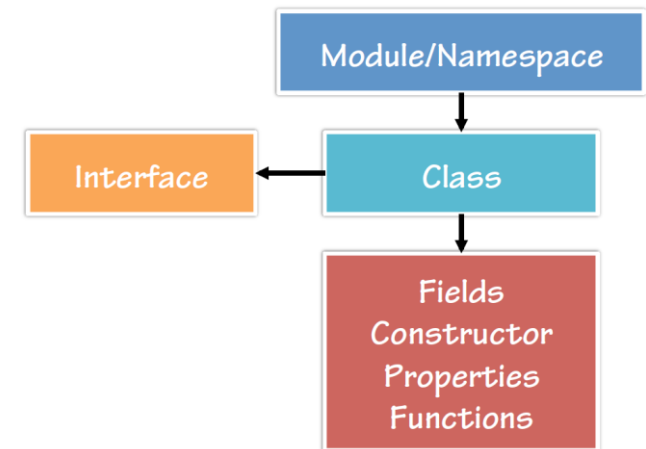
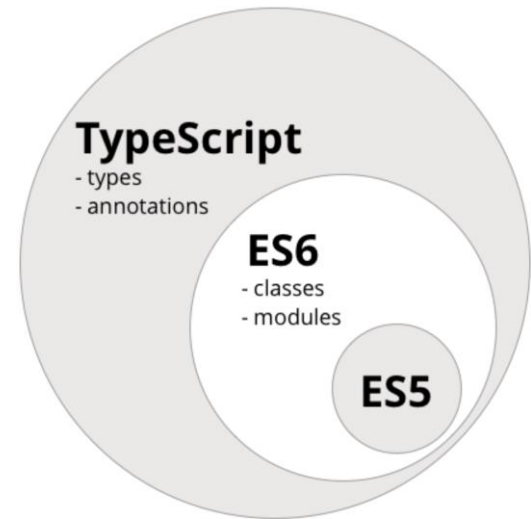
❖ Still may exist JS issues, see [Good to Know Parts When Coding with JavaScript](#), [Do's and Don't in TS](#)

❖ **Slower compilation, limited browser support, ...**

❖ **It is possible for code to pass the type checker but still throw at runtime.**

❖ **Maybe JavaScript will be enough with latest ES implementations (converging)**

❖ **BUT, [Good bye TS](#) time is approaching –Native JS [Proposals](#) (Type Annotation) [coming](#)**



Refs: [Cheat Sheets](#), [Do's&Don't](#) ,

[weird.js](#) vs. [weird.ts](#)

```
const a = null + 7; //~~~~ The value 'null' cannot be used here.
console.log(a) // 7 in JS
const b = [] + 12; //~~~~ Operator '+' cannot be applied to types 'never[]' and 'number'.
console.log(b) // 12 in JS
const ab = undefined + 7; //~~~~ The value 'undefined' cannot be used here.
console.log(ab) // NaN
```

Understand That Code Generation Is Independent of Types

> tsc .\weird.ts | node .\weird.js

tsc (TS compiler) does two things (entirely independent from each other):

- It converts next-generation TypeScript/JavaScript to an older version of JavaScript that works in browsers or other runtimes (“transpiling”). TS types are “erasable”: part of compilation to JavaScript is simply removing all the interfaces, types, and type annotations from your code.
- It checks your code for type errors. TypeScript **never** changes the **runtime behavior** of JavaScript code.

You Cannot Check TypeScript Types at Runtime (Types Erased) - e.g. Like Java Generics types erased at Runtime

```
function calculateArea(shape: Shape) {  
  if (shape.kind === 'rectangle') {  
    return shape.width * shape.height;  
  } else {  
    return shape.width * shape.width;  
  }  
}
```



```
function calculateArea(shape) {  
  if (shape.kind === 'rectangle') {  
    return shape.width * shape.height;  
  } else {  
    return shape.width * shape.width;  
  }  
}
```

Code with Type Errors Can Produce Output

```
//Code with Type Errors Can Produce Output  
let l = 'hello';  
l = 1234; //Type 'number' is not assignable to type 'string'.ts(2322)  
console.log(l) //1234
```

If you want to disable output on errors, use the **noEmitOnError** option in *tsconfig.json*, or the equivalent in your build tool or CLI.

>tsc --noEmitOnError hello.ts

Downleveling

TS has the ability to rewrite (transpile) code from newer versions of ES to older ones such as ES 3 or ES 5

From: `Hello \${person}, today is \${date.toDateString()}!`;

To: "Hello ".concat(person, ", today is ").concat(date.toDateString(), "!");

You Cannot Overload (in next sections) a Function Based on TypeScript Types

Declaring variables and constants

var, let, any(explicit, implicit), const (dedicated for constant) – CAN BE used

```
var num1 = 1; //Type inference (number)
var num3:number; //safe via Type Annotation
var num2: number = 23; //Type Annotation and the Value
//safe via type inference
num1 = "d";
```

*Type annotation,
Type inference,
both provides intelligence*

Avoid Excessive Annotations

Declaring types (explicit type annotation is redundant) **for all your variables is counterproductive** (e.g. num2).

Do it like num1 (mouse over num1, you'll see that its type has been inferred as number).

```
const person: {
  name: string;
  born: {
    where: string;
    when: string;
  };
} = {
  name: 'Alym',
  born: {
    where: 'Merv',
    when: 'c.1797',
  }
};
```

you can just write:

```
const person = {
  name: 'Alym',
  born: {
    where: 'Merv',
    when: 'c.1797',
  },
};
```

1-var_let_const.ts,
1b-type-inference.ts

Allowing types to be inferred **can also facilitate refactoring**

```
interface Product {  
  id: number;  
  name: string;  
  price: number;  
}  
  
function logProduct(product: Product) {  
  const id: number = product.id;  
  const name: string = product.name;  
  const price: number = product.price;  
  console.log(id, name, price);  
}
```

Once Change Request happened, and IDs might have letters in them in addition to numbers. So, ID must be changed.

```
interface Product {  
  id: string;  
  name: string;  
  price: number;  
}
```

Then you will have an error, because you included **explicit annotations** on all the variables

```
const id: number = product.id;  
// ~~ Type 'string' is not assignable to type 'number'
```

Better alternative - allows the types of all local variables to be inferred (using **destructuring assignment**)

```
function logProduct(product: Product) {  
  const { id, name, price } = product;  
  console.log(id, name, price);  
}
```

> tsc .\1b-type-inference.ts | node .\1b-type-inference.js

Simply, we should avoid adding type annotations literally everywhere

```
const arr: string[] = ['a', 'b'];

arr.forEach((str: string) => {
  // do something with str...
});

arr.forEach((str) => {
  // do something with str...
});
```

- ◀ Consider a simple string array
- ◀ We *could* annotate the callback parameter
- ◀ But it's completely pointless – TypeScript already knows the parameter will be a string
- ◀ We still **get full type-safety without the annotation**

Explicit type annotations are still required **in some situations** where TypeScript doesn't have enough context to determine a type on its own, e.g. function parameters (unless **callbacks** – see above example).

BUT for the local variables do no use types to define. TS will infer its type from values.

Use Different Variables for Different Types

```
let productId: string | number = "12-34-56";  
//TS has narrowed the union type based on the assignment  
fetchProduct(productId);  
productId = 123456; // OK  
fetchProductBySerialNumber(productId); // OK
```

You reused a variable This was confusing for the type checker and would be confusing for a human reader, too. Like **VAR-programming**

This is not to be confused with “shadowed” variables

```
const productId = "12-34-56";  
fetchProduct(productId);  
{  
  const productId = 123456; // OK, shadowed  
  fetchProductBySerialNumber(productId); // OK  
}
```

```
//better solution  
const productId = "12-34-56";  
fetchProduct(productId);  
const serial = 123456; // OK  
fetchProductBySerialNumber(serial); // OK
```

It disentangles two **unrelated concepts** (ID and serial number).
More specific, simpler types, no explicit type annotation.

Allows to define with **const** - **more efficient memory utilization**.

You should have far more const than let

In general, better **use different names for different concepts**. Or just disallow this sort of shadowing via linter rules such as **eslint's 'no-shadow'**

1c-types-per-variable.ts

Two ways (***: Type*** (type annotation) **vs.** ***as Type*** (type assertion)) of assigning a value to a variable and giving it a type

```
interface Person { name: string };  
  
const alice: Person = { name: 'Alice' }; //type annotation  
//type assertion (a.k.a casting, avoid this term)  
const bob = { name: 'Bob' } as Person;  
const ole = <Person> { name: 'Bob' };
```

While these **achieve similar ends**, they are **actually quite different!**

Prefer Type Annotations to Type Assertions - First verifies that the value conforms to the interface, second silences

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

```
const alice2: Person = {}; //~~~~~ Property 'name' is missing in type '{}' but required in type 'Person'  
const bob2 = {} as Person; // No error  
//The type assertion silences this error by telling the type checker that, for whatever reason, you know better than it does ;)
```

Try with adding/removing/empty property

```
const alice3: Person = {  
  name: 'Alice',  
  occupation: 'TypeScript developer' // ~~~~~~ Object literal may only specify known  
properties, and 'occupation' does not exist in type 'Person'  
};  
const bob3 = {  
  name: 'Bob',  
  occupation: 'JavaScript developer'  
} as Person; // No error - type assertion silences
```

TypeScript has an **additional tool** known as **excess property checking** that flags extra properties in objects with declared types, but it doesn't apply **if you use an assertion**.

You should **use type annotations** unless you have a **specific reason to use a type assertion**.

NOTE `<Person>` was original syntax for assertions and is equivalent to `{ } as Person`.

It is less common now because `<Person>` is interpreted as a **start tag** in **.tsx** files (TypeScript + React).

```
const bob = { name: 'Bob' } as Person; //more common  
const bobek = <Person>{name: 'Bob' }; //less used
```

[>2-prefer-annotation-to-assertion.ts](#)

So when should you use a type assertion?

When you truly do know **more about a type than TypeScript does**
e.g. browser DOM objects, or libraries with no Types,...

```
document.querySelector('#myButton')?.addEventListener('click', e => {  
  const button = e.currentTarget as HTMLButtonElement;    });  
//Because TS doesn't have access to the DOM of your page, it has no way of knowing that #myButton is a button element.
```

What if a variable's type includes null but you know from context that this isn't possible?

You can use a type assertion to remove null from a type:

```
const elNull = document.getElementById('foo'); //HTMLElement | null  
const el = document.getElementById('foo') as HTMLElement; //HTMLElement
```

This sort of type assertion is so common that it gets a special syntax and is known as a **non-null assertion**.

Suffix **!** is interpreted as a **type assertion** that the value is **non-null**.

```
const el = document.getElementById('foo')!;
```

If you're accessing a property or method on an object that might be null, use “**optional chaining**” operator, **?.**:

```
document.getElementById('foo')?.addEventListener('click', () => {  
  alert('Hi Mele Merdan!');  
});
```

> 2-prefer-annotation-to-assertion2.ts

Limitation of type assertion - Type assertions have their limits: they don't let you convert between arbitrary types.

You can't convert between a **Person** and an **HTMLElement** since their intersection is empty (i.e., the never type):

```
interface Person { name: string; }
const body = document.body;
const el = body as Person;
// ~~~~~
// Conversion of type 'HTMLElement' to type 'Person' may be a mistake because
// neither type sufficiently overlaps with the other. If this was intentional,
// convert the expression to 'unknown' first. //error fix suggestion, every type is a subtype of unknown (a universal type)
const el = document.body as unknown as Person; // OK – called force casting
```

Casting doesn't actually change the type of the data within the variable. See [TS-Casting](#)

```
let x3:unknown = '9';
console.log((x3 as number).toExponential); //undefined, since number has no length, BUT no error shown
console.log((x3 as string).length); //1
```

Not every type assertion uses the keyword **as**. See “Type guards” (**is**), which allow you to associate some logic with a type assertion to check whether it's valid.

Understand Type Narrowing - TypeScript goes from a broad type to a more specific one

There are many ways that you can narrow a type

```
const elem = document.getElementById('my-cal'); //HTMLElement | null
//narrowed type
if (elem) {
    elem.innerHTML = 'Autumn Time'.blink(); //HTMLElement
} else {
    elem // null
    alert('No element #my-cal');
}
```

```
const elem = document.getElementById('my-cal');
if (!elem) throw new Error('Unable to find #my-cal'); //check Falsiness
elem.innerHTML = 'Autumn Time'.blink();
```

```
//property check also handy
interface Apple { isGoodForBaking: boolean; }
interface Orange { numSlices: number; }
function pickFruit(fruit: Apple | Orange) {
    if ('isGoodForBaking' in fruit) {
        //Apple
    } else {
        // Orange
    }
}
```

```
//You can also use instanceof or typeof (for primitives)
function contains(text: string, search: string | RegExp) {
    if (search instanceof RegExp) {
        return !!search.exec(text);
    }
    return text.includes(search);
}
```

```
//Some built-in functions (e.g. Array.isArray) able to narrow types
function contains(text: string, terms: string | string[]) {
    const termList = Array.isArray(terms) ? terms : [terms];
}
```

Some Surprises

```
// wrong way to exclude null from a union type
const elem = document.getElementById('what-time-is-it'); //HTMLElement | null
if (typeof elem === 'object') {
    // elem: HTMLElement | null, (Remember: typeof null is Object)
}
```

//Similar surprises can come from falsy primitive values

//Empty string and 0 are both falsy, x could still be a string or number in that branch.

```
function maybeLogX(x?: number | string | null) {
    if (!x) {
        console.log(x);
        // ^? (parameter) x: string | number | null | undefined
    }
}
```

Another way to narrow your types is by putting an explicit **“tag”** on them – a.k.a **“tagged union”** or **“discriminated union”**

```
interface UploadEvent
{ type: 'upload'; ... }
interface DownloadEvent
{ type: 'download'; ..}
type AppEvent =
    UploadEvent | DownloadEvent;
```

Try also !! - Type guards to narrow types

```
function handleEvent(e: AppEvent) {
    switch (e.type) {
        case 'download':
            console.log('Download', e.filename);
            break;
        case 'upload':
            console.log('Upload', e.filename, e.contents.length, 'bytes');
            break;
    }
}
```

Limit Use of the any Type

TypeScript's type system:

Gradual - you can add types to your code bit by bit (with **noImplicitAny**).

Optional – when *type checker* disabled (silenced).

The key to these features is the **any** type:

➤ there's No Type Safety with any

```
let ageInYears: number;
ageInYears = '12';
// ~~~~~ Type 'string' is not assignable to type 'number'.
ageInYears = '12' as any; // OK – silenced by type assertion 'as any'

//calc age after one year - chaos
ageInYears += 1; // OK; at runtime, ageInYears is now "121"
```

➤ breaks contracts

```
function calculateAge(birthDate: Date): number {
    //TBD
    return 1; //tbd
}

let birthDate: any = '19e90-01-19'; // let you break the contract
calculateAge(birthDate); // OK
```

Using any eliminates many
of the advantages of using
TypeScript !!!

>tsc.\2b-limit-any.ts | node .\2b-limit-any.js

➤ There Are No Language Services (autocomplete and contextual doc) for any Types

```
let person1 = {name: "Oli", age:12};
person1.
  age (property) age: number
  name
let person2: any = {name: "Oli", age:12};
person2.
```

```
let msg; //implicit ANY, no INTELLISENSE
//bad experience, like repeating VAR experience
msg = false; //no INTELLISENSE
msg = 123; //no INTELLISENSE
msg = "abc"; //no INTELLISENSE

//1-way cast
(<string>msg).startsWith("a");
(<number>msg).toPrecision;
//2-way
(msg as string).startsWith("a");
```

How to get **intellisense** support with **ANY** or **Unknown**?
Use **Type assertion(Casting)** which gives context aware smartness.

More reasons:

- any Types Mask Bugs When You Refactor Code
- any Hides Your Type Design (e.g. state)
- any Undermines Confidence in the Type System (expect many errors at runtime)

Avoid using ANY as you can!!!
But use it if you feel SAFE.

[2-z-limit-any.ts](#)
[2-data-types.ts](#)

Use unknown Instead of any for Values with an Unknown Type

```
function parseYAML(yaml: string): any {
    // ...
}
interface Book {
    name: string;
    author: string;
}
const book: Book = parseYAML(`
    name: Wuthering Heights
    author: Emily Brontë
`);
console.log(book.title); // YES error
```

```
//Without the type annotation, though, the book variable would quietly
//get an any type, thwarting type checking wherever it's used:
const book = parseYAML(`
    name: Jane Eyre
    author: Charlotte Brontë
`);
console.log(book.title); // No error, logs "undefined" at runtime
book('read'); // No error, throws "book is not a function" at runtime
```

Type: unknown

unknown is a similar, but **safer alternative to any**. TypeScript will prevent **unknown** types from being used.

```
function safeParseYAML(yaml: string): unknown {
    return parseYAML(yaml);
}
```

```
const book = safeParseYAML(`
    name: The Tenant of Wildfell Hall
    author: Anne Brontë
`);
console.log(book.title); //ERROR 'book' is of type 'unknown'
book("read"); // Error: 'book' is of type 'unknown'

console.log((book3 as Book).title); //Property 'title' does not exist on type 'Book'.
console.log((book3 as Book).author);
```

unknown can also be used instead of **any** in “double assertions”:

```
declare const foo: Foo;
let barAny = foo as any as Bar;
let barUnk = foo as unknown as Bar;
```

To understand the unknown type, it helps to think about any in terms of assignability. **The power and danger of any:**

- All types are assignable to the any type.
- The any type is assignable to all other types.

>tsc .\2b-use-unknown.ts

Config Best Practices — Know Which Compiler Options are Used? See: [tsconfig.json](#)



tsconfig.json

To enable all of these checks, turn on the strict setting.
Without strict mode, you lose TS usefulness.

- Always enable additional cleanliness/type-checking options:
 - `noImplicitOverride`
 - `noImplicitReturns`
 - `noPropertyAccessFromIndexSignature`
 - `noUncheckedIndexedAccess`
 - `noFallThroughCasesInSwitch`
 - `noUnusedLocals`
 - `noUnusedParameters`
 - `exactOptionalPropertyTypes`
- Always disable the `allowUnreachableCode` option
- Always configure an `outDir`
- Always set `noEmitOnError`

Strict mode is an umbrella for:

- `alwaysStrict`
- `strictNullChecks`
- `strictBindCallApply`
- `strictFunctionTypes`
- `strictPropertyInitialization`
- `noImplicitAny`
- `noImplicitThis`
- `useUnknownInCatchVariables`

Stricter than strict (aggressive TS) options e.g. `noUncheckedIndexedAccess`

```
//no type err under --strict opt., but throws an EX at runtime:
const tensez = ['past', 'present', 'future'];
tensez[3].toUpperCase();
//With noUncheckedIndexedAccess set, this is an
error:tenses[3].toUpperCase(); // ~~~~~ Object is possibly
'undefined'.
//However this come with cost. Many valid accesses will also be
flagged as possibly undefined:
tenses[0].toUpperCase(); // ~~~~~ Object is possibly 'undefined'.
```

If a teammate shares an issue and you're unable to reproduce
his errors, make sure your
compiler options are the same !!!!

When appropriate to use aggressive check? Just be aware.

noImplicitAny controls what TypeScript does when it **can't determine** the type of a variable.

```
//Does this code pass the type checker? Yes | No
function add(a, b) {
    return a + b;
}
//hover on function, inferred as
//function add(a: any, b: any): any
add(10, null); //10
```

You can set flag via command line or in `tsconfig.json`

`>tsc --noImplicitAny app.ts`

This will help TypeScript spot problems, improve the readability of your code, and enhance your development experience when TS has type information.

When appropriate to set **noImplicitAny** off?

strictNullChecks controls whether **null** and **undefined** are permissible values in every type. Helpful to catch errors.

```
//This code is valid when strictNullChecks is off:
const x1: number = null; // OK, null is a valid number
const x2: number = undefined; // OK, undefined is a valid number

// ~ Type 'null'/'undefined' is not assignable to type 'number' when strictNullChecks is ON

const x3: number | null = null; //even with strictNullChecks is ON
```

You should certainly set **noImplicitAny** before you set **strictNullChecks**

If you **choose to work without strictNullChecks**, use explicit checks

```
const statusEl = document.getElementById('status');
// ~~~~ 'statusEl' is possibly 'null'.
if (statusEl) {
    statusEl.textContent = 'Ready'; // if statement in this way is known as “narrowing” or “refining” a type
}
```

When appropriate to set **strictNullChecks** off?

3-config-options.ts

Null Assertion (!)

TypeScript's inference system isn't perfect, there are times when it makes sense to ignore a value's possibility of being **null** or **undefined**.

```
statusEl!.textContent = 'Ready'; // use !" which is a "non-null assertion."
```

Just like casting, this '!' can be unsafe and should be used with care.

Nullish coalescing assignment operator (??=) is used between two values. If the first value is **undefined** or **null**, the second value is assigned.

The **??=** operator is an [ES2020 feature](#).

```
let x;  
x ??= 5;
```

```
let name = null;  
let text = "missing";  
let result = name ?? text;
```

In JAVA //ternary operator
String result = name != null ? name: "unknown";
//Using **Elvis (as his hair) Operator** in Java Spring
result = name?:"unknown";

```
function printMileage(mileage: number | null | undefined) {  
  console.log(`Mileage: ${mileage ?? 'Not Available'}`);  
}
```

```
printMileage(null); // Prints 'Mileage: Not Available'  
printMileage(0); // Prints 'Mileage: 0'
```

The Optional Chaining Operator (?.)

The **?.** operator returns **undefined** if an object is **undefined** or **null** (instead of throwing an error). E.g. accessing deeply nested properties can lead to errors

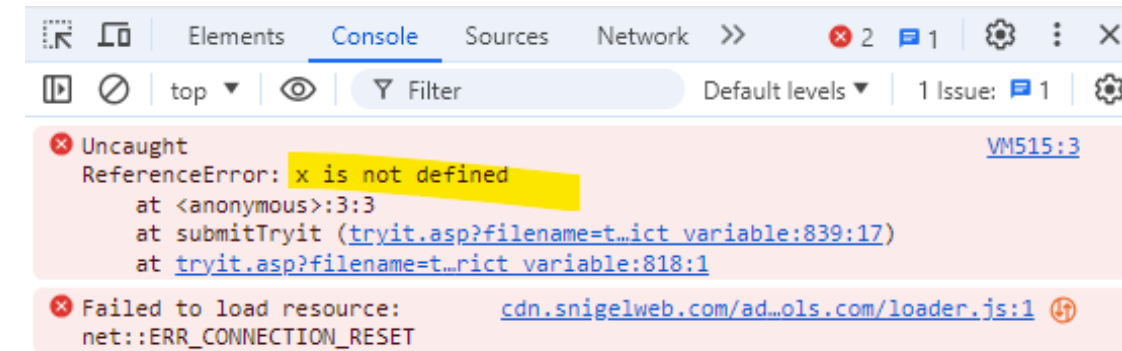
```
const car = {type:"Fiat", model:"500", color:"white"};  
document.getElementById("demo").innerHTML = car?.name;
```

```
//no explicit check needed  
const user = { profile: { name: 'Jane' } };  
const userName = user?.profile?.name;  
console.log(userName); // "Jane"
```

JavaScript Use Strict

"**use strict**"; Defines that JavaScript code should be executed in "strict mode"
e.g. JavaScript in strict mode **does not allow variables to be used if they are not declared**. Fixes hoisting misuse,

```
"use strict";  
x = 3.14;      // cause an error because x is not declared  
myFunction();  
  
function myFunction() {  
  y = 3.14;    // Also cause an error because y is not declared  
}
```



Declared inside a function, it has local scope (only the code inside the function is in strict mode):

```
x = 3.14; // This will not cause an error.  
myFunction();  
  
function myFunction() {  
  "use strict";  
  y = 3.14; // This will cause an error  
}
```

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript. It changes previously accepted (**tolerated**) "bad syntax" into real errors.

Mistyping a variable name creates a new global variable. In strict mode, this will throw an error

No error or feedback assigning values to non-writable props. in loose mode. In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

```
"use strict";
x = {p1:10, p2:20};
//Using an object(variables), without declaring it, is not allowed
```

```
"use strict";
let x = 3.14;
delete x; //Deleting a variable (or object) is not allowed.
```

```
"use strict";
function x(p1, p2) {};
delete x; // Deleting a function is not allowed.
```

```
"use strict";
function x(p1, p1) {}; // Duplicating a parameter name is not allowed
```

```
"use strict";
let x = 010; // Octal numeric literals are not allowed
```

```
"use strict";
const obj = {};
Object.defineProperty(obj, "x", {value:0, writable:false});

obj.x = 3.14; // Writing to a read-only property is not allowed
```

```
Uncaught
ReferenceError: x is not defined
    at <anonymous>:3:3
    at submitTryit (tryit.asp?filename=t...trict object:839:17)
    at tryit.asp?filename=t...strict object:818:1
Failed to load resource: net::ERR_CONNECTION_RESET loader.js:1
```

```
Uncaught
SyntaxError: Delete of an unqualified identifier in strict mode.
    at submitTryit (tryit.asp?filename=t...trict delete:840:17)
    at tryit.asp?filename=t...strict delete:819:1
Failed to load resource: cdn.snigelweb.com/ad...ols.com/loader.js:1
net::ERR_CONNECTION_RESET
```

```
Uncaught
SyntaxError: Duplicate parameter name not allowed in this context
    at submitTryit (tryit.asp?filename=t...ct duplicate:839:17)
    at tryit.asp?filename=t...ict duplicate:818:1
Failed to load resource: net::ERR_CONNECTION_RESET loader.js:1
```

```
Uncaught
SyntaxError: Octal literals are not allowed in strict mode.
    at submitTryit (tryit.asp?filename=t...strict octal:839:17)
    at tryit.asp?filename=t... strict octal:818:1
Failed to load resource: cdn.snigelweb.com/ad...ols.com/loader.js:1
net::ERR_CONNECTION_RESET
```

```
Uncaught
TypeError: Cannot assign to read only property 'x' of object '#<Object>'
    at <anonymous>:6:7
    at submitTryit (tryit.asp?filename=t...ict readonly:842:17)
    at tryit.asp?filename=t...rict readonly:821:1
Failed to load resource: net::ERR_CONNECTION_RESET loader.js:1
```

More about strict rules [examples](#)

Note: The **this** keyword in functions behaves differently in strict mode. See part-1 demo ...

Think of Types as Sets of Values

3-string-literal-types-type-aliases.ts

In TypeScript a variable just has a type. This is best thought of as a set of possible values (e.g. number types has 12, -3, ..). The smallest set is the **empty set** - contains no values. It corresponds to the **never type(or bottom type)**

Type: never - never effectively throws an error whenever it is defined.

```
let x: never = 12; // Error: Type 'number' is not assignable to type 'never'.
let y: never = true; // Error: Type 'boolean' is not assignable to type 'never'.
```

String Literal Types

The next **smallest sets** are those that contain single values. These correspond to **literal types** in TypeScript.

```
//string literals - acts like distinct types,
//e.g. yes|no, no need explicit validation etc.
let mr : 'Mister'; //type of variable
let mr1: 'Mister' = 'Mister';
let mr2: 'Mister' = 'Madam'; // err - type is not assignable

//To form types with two or three values, you can union literal types
//gives enum like behavior, e.g. to finite values
let mr3: 'Mister' | 'Madam' = 'Madam';
let mr4: 'Mister' | 'Madam' = 'Miss'; //err
```

Table 2-1. TypeScript terms and set terms

TypeScript term	Set term
never	∅ (empty set)
Literal type	Single element set
Value assignable to T	Value ∈ T (member of)
T1 assignable to T2	T1 ⊆ T2 (subset of)
T1 extends T2	T1 ⊆ T2 (subset of)
T1 T2	T1 ∪ T2 (union)
T1 & T2	T1 ∩ T2 (intersection)
unknown	Universal set

Type aliases [only in TS]
(shorthand way of string literal types)

```
let mr5: 'Mister' | 'Madam' | 'Miss' = 'Miss';
type mrCategory = 'Mister' | 'Madam' | 'Miss';
let mr6: mrCategory= 'Mister';
let mr7: mrCategory= 'Madam';
let mr8: mrCategory= 'Mrs'; // err - is not assignable to type
```

Use Never Types to Perform Exhaustiveness Checking

For a switch statement to be exhaustive, it just needs to check every possible value of the variable being switched on. The `never` type is used to express a value that should never happen. A function that throws an error, so never returns, can have an inferred return type of `never`

```
type Coord = [x: number, y: number];
interface Box {
  type: 'box';
  topLeft: Coord;
  size: Coord;
}
interface Circle {
  type: 'circle';
  center: Coord;
  radius: number;
}
type Shape = Box | Circle;
```

```
//You can draw these using built-in canvas methods:
function drawShape(shape: Shape, context: CanvasRenderingContext2D) {
  switch (shape.type) {
    case 'box':
      //TDD
    case 'circle':
      //TBD
  }
}
```

```
//Now add a third shape:
interface Line {
  type: 'line';
  start: Coord;
  end: Coord;
}
```

`type Shape = Box | Circle | Line;`

There are no type errors, but this change has introduced a bug: drawShape will silently ignore any line shapes (error of omission).

No value is assignable to the never type, and we can use this to **turn an omission into a type error**:

```
function assertUnreachable(value: never): never {
  throw new Error(`Missed a case! ${value}`);
}
```

```
switch (shape.type) {
  case 'box':
    context.rect( ... shape.topLeft, ... shape.size);
    break;
  case 'circle':
    context.arc( ... sha
    break;
  default:
    assertUnreachable(shape);
}
```

Argument of type 'Line' is not assignable to parameter of type 'never' (parameter) shape: Line
View Problem (Alt+F8) No quick fixes available

Basic (and Object) Data Types: boolean, number, string, **any** (avoid it **no typing benefit**), and ES 2015 new primitive type **'symbol'**

Custom: enum, classes, interfaces, array, ...

//e.g. Cobol Punch Card has 5 fields on **explicit** positions/indices 0-6, 7, 8-11, 12-72, 72-80

```
enum PunchCard { Sequence = 0, Indicator = 7, AreaA = 8, AreaB = 12, IdentificationArea = 73 };
```

```
let startPosition: PunchCard = PunchCard.AreaA;
console.log(startPosition); //8
console.log(PunchCard.IdentificationArea); //73
let fieldName: string = PunchCard[startPosition];
console.log(fieldName); //AreaA
console.log(PunchCard[12]); //AreaB
```

By default enum uses auto-initializing
zero based indices

//Also enums can be used with string values, mix, ...

```
enum PrintMedia { Newspaper = "NEWSPAPER", Magazine = "MAGAZINE"}
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE
```

```
type Strs = 'a' | 'b' | 'c';
```

```
enum Str {
  A = 'a',
  B = 'b',
  C = 'c',
}
```

◀ Enums are **similar in some ways to union types**
and can be used in similar contexts sometimes

How enums in TS it behaves when
mapping to DATABASE???

Prefer More Precise Alternatives to String Types

Domain of a type is the set of values assignable to that type

```
interface Album {
  artist: string;
  title: string;
  releaseDate: string; // YYYY-MM-DD
  recordingType: string; // E.g., "live" or "studio"
}

const kindOfBlue: Album = {
  artist: 'Miles Davis',
  title: 'Kind of Blue',
  releaseDate: 'August 17th, 1959', // Oops!
  recordingType: 'Studio', // Oops!
};
```

With narrowed types, TypeScript can do its best to check for errors

Can you make the types narrower to prevent these sorts of issues?

```
type RecordingType = 'studio' | 'live';
interface Album2 {
  artist: string;
  title: string;
  releaseDate: Date;
  recordingType: RecordingType;
}

const kindOfBlue2: Album2 = {
  artist: 'Miles Davis',
  title: 'Kind of Blue',
  releaseDate: new Date('1959-08-17'),
  recordingType: 'Studio'
  // ~~~~~ Type '"Studio"' is not assignable to type 'RecordingType'
};
```

- Avoid “stringly typed” code. Prefer more appropriate types
- Prefer a union of string literal types to string if that more accurately describes the domain of a variable. You’ll get stricter type checking and improve the development experience.
- Prefer **keyof T** to string for function parameters that are expected to be properties of an object.

//Symbols

New EcmaScript 2015 feature, in tsconfig.json change compiler option: "target": "ES2015" (no ES5).

No **new** used to create symbol. No constructor function. Symbols are new primitive data type. Its type is SYMBOL. It is unique

```
var Symbol: SymbolConstructor
(description?: string | number) => symbol

Returns a new unique Symbol value.

@param description — Description of the new Symbol object.

let mysym = Symbol()
```

```
let mySym1 = Symbol(23);
let mySym2 = Symbol(23);
console.log(mySym1 === mySym2); //false
console.log(typeof mySym2); //symbol
```

//Arrays

```
let arr1: string[] = ['a', 'b', 'c'];
let arr2: Array<string> = ['a', 'b', 'c'];
let arr3: any[] = ['a', true, 23];

const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]
const evens = numbers.filter(num => num % 2 === 0);
console.log(evens); // [2, 4]
const sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // 15
```

Type Inference TypeScript can infer the type of an array if it has values.

See also, [Typed Arrays](#) (Int8Array, Uint32Array, Float64Array, ..)

Problem: Iterating over arrays to transform, filter, or accumulate values can be repetitive.

Use ``map()``, ``filter()``, and ``reduce()`` for common array operations – provides **functional approach** to transforming, filtering, and reducing arrays, making your code more expressive and concise.

JAVA 8 way

More methods:

``find()`` / ``findIndex()``, ``some()`` / ``every()``, ``flat()`` ?
``flatMap()``, ``from()`` / ``of()``,

```
let f = [] //let f: never[]
f[5] = "some value"
//not assignable to type 'never'
```

[3-symbol.ts](#), [3-arrays-tuples.ts](#)

//Tuple

A **tuple** is a typed [array](#) with a **pre-defined length (fixed array)** and types for each index.

```
let tuple: [number, string] = [123, "Broadcom"];
```

Order matters in tuple and will throw an error.

```
let tuple2: [number, string] = ["Broadcom", 123];
```

```
//prog.ts(4,34): error TS2322: Type 'string' is not assignable to type 'number'.
```

// define tuple & initialize correctly

```
let ourTuple: [number, boolean, string];
```

```
ourTuple = [5, false, 'Coding Bear was here']; // We have no type safety in our tuple for indexes starting from 3+
```

```
ourTuple[3] = 'Something new and wrong'; //compiler ERROR, BUT you can by-pass with push()
```

```
ourTuple.push('Something new and wrong'); //no control mechanism yet
```

```
ourTuple.push(34);
```

```
console.log(ourTuple);//[ 5, false, 'Coding Bear was here', 'Something new and wrong', 34 ]
```

// instead use **readonly** tuple

```
const ourReadOnlyTuple: readonly [number, boolean, string] = [5, true, 'The Real Coding God'];
```

```
// shows error as it is readonly - ONLY COMPILE TIME, no type safety in TUPLE for indexes starting from N+
```

```
ourReadOnlyTuple.push('Coding day off');
```

YOU can add elements to tuples with PUSH, it spoils its structure

> tsc 3-arrays-tuples.ts | node .\3-arrays-tuples.js

Tuples are simply fixed length arrays, where each element has a specific type.

Tuples can be used in the same way as arrays, because they are arrays!

```
type Coords = [number, number];
```

```
type Coords = number[];
```

```
type CoordsAndName = [  
  number,  
  number,  
  string,  
];
```

- ◀ A mixture of types can be specified
- ◀ The type for each element goes inside the square brackets
- ◀ With a regular array there are no limits to the number of elements in the array
- ◀ **Only number types are allowed**
- ◀ With tuples, we can specify an array that contains **mixed types**

Thinking of types as sets can also clarify the relationships between arrays and tuples.

```
const list = [1, 2];  
// ^? const list: number[]  
const tuple: [number, number] = list;  
// ~~~~ Type 'number[]' is not assignable to  
// type '[number, number]'  
// Target requires 2 element(s) but source may have  
// fewer. The empty list and the list [1] are examples
```

Tuple is fixed array BUT it's **not a subset** of it.

Is a triple assignable to a pair? Via structural typing?

```
const triple: [number, number, number] = [1, 2, 3];  
const double: [number, number] = triple;  
// ~~~~~ '[number, number, number]' is not assignable to '[number, number]'  
// Source has 3 element(s) but target allows only 2.
```

Getting lat/long coordinates mixed up could be catastrophic! – Use NAMED Tuple

```
type Coords = [  
  lat: number,  
  lng: number,  
];
```

- ◀ We can label the elements in a tuple to better communicate the meaning of each element
- ◀ This labelling gives us much better intellisense and type-safety
- ◀ This is called a **named tuple**

```
type Range = [  
  start: number,  
  end: number  
];
```

> tsc 3-arrays-tuples.ts

Structural Typing (assignable to)

T1 assignable to T2 $T1 \subseteq T2$ (subset of)

If it walks like a duck and talks like a duck, then it probably is a duck.

TypeScript models this behavior using what's known as a **structural type system**.

JavaScript encourages “**duck typing**” - if you pass a function a value with all the right properties, it won't care how you made the value.

No relation needed between types.

```
interface Vector2D {
  x: number;
  y: number;
}

function calculateLength(v: Vector2D) {
  return Math.sqrt(v.x ** 2 + v.y ** 2);
}
```

```
interface NamedVector {
  name: string;
  x: number;
  y: number;
}

const v: NamedVector = { x: 3, y: 4, name: 'Pythagoras' };
const result = calculateLength(v); // OK, result is 5
```

But this (*structures compatibility*) can also lead to trouble.

```
const n = normalize({ x: 3, y: 4, z: 5 });
console.log(n); // { x: 0.6, y: 0.8, z: 1 }
```

```
function normalize(v: Vector3D) {
  const length = calculateLength(v);
  return {
    x: v.x / length,
    y: v.y / length,
    z: v.z / length,
  };
}
```

If you want this to be an error, you have some options:

- 1) Use an optional never type in your interface to disallow a property (prohibit Z).
- 2) Or use branded types (next slide).

>tsc .\3-structural-types.ts | node .\3-structural-types.js

Combining Types (union, intersection)

Thinking of types as sets of values helps you reason about operations on them.

$T1 \mid T2$	$T1 \cup T2$ (union)
$T1 \& T2$	$T1 \cap T2$ (intersection)

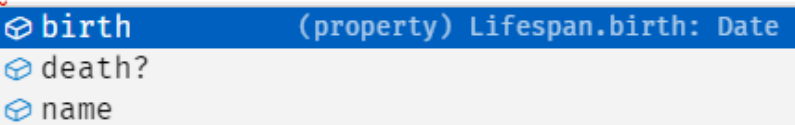
```
//union types
function getReport1(id: number | string) {
    //TBD
}
```

```
//intersection types - like declaration merging,.
function getReport2(id: Mobile & Tablet) {
    //TBD
}
```

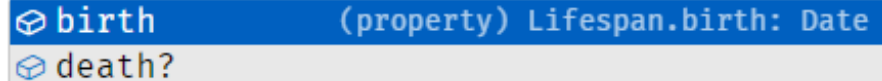
TypeScript types form intersecting sets (a Venn diagram) rather than a strict hierarchy. Two types can overlap without either being a subtype of the other. Intersection acts like declaration merging (union of properties) – see next sections.

```
interface Person {
    name: string;
}
interface Lifespan {
    birth: Date;
    death?: Date;
}
```

```
type PersonSpan = Person & Lifespan;
const p: PersonSpan = {name: 'Alan', birth: new Date()};
p.
```



```
type PersonUnion = Person | Lifespan;
const u: PersonUnion = {birth: new Date()};
u.
```



```
type K = keyof (Person | Lifespan);
// ^? type K = never
```

```
// Just (math) relationships, not TypeScript code!
keyof (A&B) = (keyof A) | (keyof B)
keyof (A|B) = (keyof A) & (keyof B)
```

More idiomatic (natural) way to write the PersonSpan type is via **extends**, see next page!!!

3-combining-types.ts

extends?

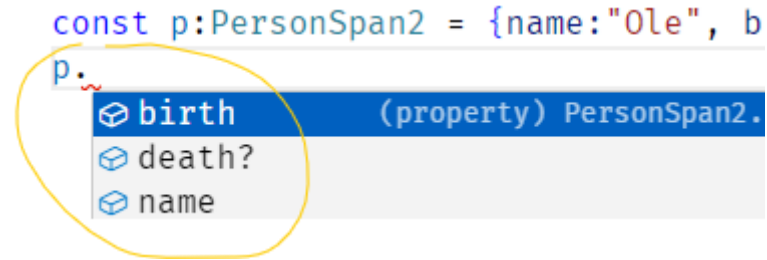
T1 extends T2

T1 \subseteq T2 (subset of)

Thinking of types as sets of values, **what does extends mean?**
Just like “**assignable to**”, you can read it as “**subset of**.”

```
interface Person {
  name: string;
}

interface PersonSpan extends Person {
  birth: Date;
  death?: Date;
}
```



More nuanced type relationships

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

interface Student extends NullableStudent {
  ageYears: number;
}
```

If you try to expand the type of `ageYears` instead, you'll get an error:

Not every language would let you change the type of `ageYears` like this.

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

```
interface StringyStudent extends NullableStudent {
  // ~~~~~
  // Interface 'StringyStudent' incorrectly extends
  interface 'NullyStudent'.
  ageYears: number | string;
}
```

Think of “extends,” “assignable to,” and “subtype of” as synonyms for “subset of.”

3-extends.ts



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>