Part4 - 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)

Use Type Operations and Generic Types to Avoid Reparative (Duplicated) Code

DRY (Don't Repeat Yourself) is a one of element in SOLID Principle

6-avoid-repetative-code.ts

e.g. Extremely repetitive (duplicated) code - same line was copied and pasted/modified, calculation is wrong, .. .

```
console.log(
    'Cylinder r=1 * h=1',
    'Surface area:', 6.283185 * 1 * 1 + 6.283185 * 1 * 1,
    'Volume:', 3.14159 * 1 * 1 * 1
);
console.log(
    'Cylinder r=1 * h=2',
    'Surface area:', 6.283185 * 1 * 1 + 6.283185 * 2 * 1,
    'Volume:', 3.14159 * 1 * 2 * 1
);
console.log(
    'Cylinder r=2 * h=1',
    'Surface area:', 6.283185 * 2 * 1 + 6.283185 * 2 * 1,
    'Volume:', 3.14159 * 2 * 2 * 1
);
```

Using DRY principle via factoring out some functions, shared constant, and a loop

```
type CylinderFunction = (r: number, h: number) => number;
const surfaceArea: CylinderFunction = (r, h) => 2 * Math.PI * r
* (r + h);
const volume: CylinderFunction = (r, h) => Math.PI * r * r * h;
for (const [r, h] of [[1, 1], [1, 2], [2, 1]]) {
    console.log(
        `Cylinder r=${r} * h=${h}`,
        `Surface area: ${surfaceArea(r, h)}`,
        `Volume: ${volume(r, h)}`);
}
```

E.g. function with repetitive parameters

```
function distance(
     a: { x: number, y: number },
     b: { x: number, y: number })
{
    return Math.sqrt((a.x - b.x) ** 2 + (a.y - b.y) ** 2);
}
```

Simplest way to reduce repetition is by naming your types

```
interface Point2D {
    x: number;
    y: number;
}
function distance(a: Point2D, b: Point2D) { /* ... */ }
```

Generally developers avoid duplication in code - but not **in types**. Duplication in types has many of the same problems as duplication in code. Eliminate the repetition by making one interface extend the other

```
interface Person {
    firstName: string;
    lastName: string;
}
interface PersonWithBirthDate {
    firstName: string;
    lastName: string;
    birth: Date;
}
```

```
interface Person {
    firstName: string;
    lastName: string;
}
interface PersonWithBirthDate extends Person {
    birth: Date;
}
```

You can also **use the intersection operator (&) to extend an existing type**. This technique is most useful when you want to add some additional properties to a union type

```
type PersonWithBirthDate = Person & { birth: Date };
```

Duplicated types aren't always so easy to spot. Sometimes they can be obscured by syntax

```
If several functions share the same type signature, for instance
function get(url: string, opts: Options): Promise<Response> { /* ... */ }
function post(url: string, opts: Options): Promise<Response> { /* ... */ }
```

```
Then factor out a named type for the signatures
type HTTPFunction = (url: string, opts: Options) => Promise<Response>;
const get: HTTPFunction = (url, opts) => { /* ... */ };
const post: HTTPFunction = (url, opts) => { /* ... */ };
```

6-avoid-repetative-code.ts

You can also go the other direction (**not add property but remove or use subset**). What if you have a type, State, which represents the state of an entire application, and another, TopNavState for internal use.

```
interface State {
    userId: string;
    pageTitle: string;
    recentFiles:
string[];
    pageContents: string;
}
interface TopNavState {
    userId: string;
    pageTitle: string;
    recentFiles:
string[];
    // omits pageContents
}
```

You can remove duplication in the types of the properties by **indexing into State:**

```
interface TopNavState {
    userId: State['userId'];
    pageTitle: State['pageTitle'];
    recentFiles: State['recentFiles'];
};
```

Above solution is still repetitive. You can do better with a mapped type.
Mouse over TopNavState, you see exactly the same as the previous one

```
type TopNavState = {
    [K in 'userId' | 'pageTitle' | 'recentFiles']: State[K]
};
```

Mapped types are the type system equivalent of looping over the fields in an array. **This particular pattern** is so common that it's part of the standard library, where it's called **Pick**

```
type Pick<T, K> = { [k in K]: T[k] };
```

So, **you will like this.** Just hover-over, you will see the exactly the same

```
type TopNavState = Pick<State, 'userId' | 'pageTitle' | 'recentFiles'>;
```

6-avoid-repetative-code.ts

Another form of duplication can arise with tagged unions

```ts
interface SaveAction {
type: 'save';
// ...
}
interface LoadAction {
    type: 'load';
// ...
}
type Action = SaveAction | LoadAction;
type ActionType = 'save' | 'load'; // duplicated types
```

Just define ActionType without repeating yourself by indexing into the Action union. Just hover-over, you will see the exactly the same

```ts
type ActionType = Action['type']; //try 'typez'
```

When you add more types to the **Action union**, ActionType will react automatically. **This type is distinct** from what you'd get using **Pick** – this gives you an interface with a type property

```ts
type ActionRecord = Pick<Action, 'type'>;
```

Just hover-over both to see the difference.

Pick gives you an interface(type) with a type property

```ts
type ActionType = "save" | "load"
```

```ts
type ActionRecord = {
    type: "save" | "load";
}
```

6-avoid-repetative-code.ts

keyof with explicit keys

keyof is a keyword in TypeScript which is used to extract the key type from an object type.
When used on an object type with explicit keys, keyof creates a **union type** with those keys.

```
interface Person {   name: string;   age: number; }
// `keyof Person` creates a union type of "name" and "age", equivalent to: "name" | "age"
```

```
interface Options {
    width: number;
    height: number;
    color: string;
    label: string;
}
```

```
interface OptionsUpdate {
    width?: number;
    height?: number;
    color?: string;
    label?: string;
}
```

```
// class can be initialized and later updated, & update method params might optionally include most of the same parameters (duplication) as the constructor
class UIWidget {
    constructor(init: Options) { /* ... */ }
    update(options: OptionsUpdate) { /* ... */ }
}
```

You can construct **OptionsUpdate** from Options using a **mapped type and keyof**. The **?** makes each property optional

```
type OptionsUpdate = { [k in keyof Options]?: Options[k] }; (*)
class UIWidget {
    constructor(init: Options) { /* ... */ }
    update(options: OptionsUpdate) { /* ... */ }
}
```

Above (*) pattern is also very common and is included in the standard library as **Partial utility class**

```
class UIWidget {
    constructor(init: Options) { /* ... */ }
    update(options: Partial<Options>) { /* ... */ }
}
```

6-avoid-repetative-code.ts

Generics

Generics makes it easier to write **reusable code**.

Functions

Generics with functions help make more generalized methods which more accurately represent the types used and returned. TypeScript can also infer the type of the generic parameter from the function parameters

```
function createPair<S, T>(v1: S, v2: T): [S, T] {
    return [v1, v2];
}
console.log(createPair<string, number>('hello', 42));// ['hello', 42]
console.log(createPair<number, string>(63, 'ola')); // [63, 'ola']
```

Classes

Generics can be used to create generalized classes, like Map. TS can also infer the type of the generic parameter if it's used in a constructor parameter.

```
class NamedValue<T> {
  private _value: T | undefined;
  constructor(private name: string) {}

  public setValue(value: T) {
    this._value = value;
  }
  public getValue(): T | undefined {
    return this._value;
  }
  public toString(): string {
    return `${this.name}: ${this._value}`;
  }
}

let value = new NamedValue<number>('myNumber');
value.setValue(10);
console.log(value.toString()); // myNumber: 10
```

Generics allow us to create flexible types with Type Aliases

Generics in type aliases allow creating types that are more reusable.

```
type Wrapped<T> = { value: T };
const wrappedValue: Wrapped<number> = { value:10};
```

This also works with interfaces as well: `interface Wrapped<T> {}`

All of the **built-in utility types are generics**, so we can use them for our own types:

```
type newType = Partial<MyOwnType>
```

6-avoid-repetative-code.ts

Default Value

Generics can be assigned default values which apply if no other value is specified or inferred.

Extends

Constraints can be added to generics **to limit what's allowed**. The constraints make it possible to rely on a more specific type when using the generic type. This can be combined with a default value.

```
function createLoggedPair<S extends string | number,
T extends string | number>(v1: S, v2: T): [S, T] {
  console.log(`creating pair: v1='${v1}', v2='${v2}'`);
  return [v1, v2];
}
```

Like JAVA sealed classes

```
class NamedValue<T = string> {
  private _value: T | undefined;

  constructor(private name: string) {}

  public setValue(value: T) {
    this._value = value;
  }

  public getValue(): T | undefined {
    return this._value;
  }

  public toString(): string {
    return `${this.name}:
${this._value}`;
  }
}

let value = new NamedValue('myNumber');
value.setValue('myValue');
console.log(value.toString()); //
myNumber: myValue
```

Use Conditional Types

Conditional types are like ternary statements but for types

```
type NewType =
  TestType extends ReferenceType
    ? OutputTypeA
    : OutputTypeB
```

- If TestType is assignable to ReferenceType
- Return OutputTypeA
- Otherwise return OutputTypeB

6-avoid-repetative-code.ts

Improving code readability

Loops in JS: `for`, `while`, `do/while`

But what if we want to do something with all items in an array or object?

```
let myArray = [ "banana", "pineapples", "strawberries" ]
console.log(`I have 1 ${myArray[0]}`)
console.log(`I have 2 ${myArray[1]}`) … becomes unmanageable once bigger, so use Iterable.
```

Arrays, strings, maps and sets **are iterable**, but **objects are not**.

Iteration. Iterable data can have their values fully extracted with the **for...of** or **for...in** loop, which will assign each element of an iterable to a variable.

```
let x = [ "lightning", "apple", "squid", "speaker" ]
for(let item of x) { //loops via values
console.log(item) //lightning, apple, squid, speaker
}
```

```
let x = [ "lightning", "apple", "squid", "speaker" ]
for(let item in x) {//loops via properties/keys/indexes
console.log(item) //0, 1, 2, 3
}
```

for...of and for...in differ in the way **they handle undefined values**

```
let x = [] //let f: never[]
x[5] = "some value"
for(let item of x) {
console.log(item)
}
//undefined, undefined, undefined, undefined, undefined, "some value"
```

With **for...in**, we only get indices and not array values, so it therefore omits any undefined values:

```
for(let item in x) {
console.log(item)
} // Will show: 5
```

6-code-readability.ts

Do not use **for in** over an Array if the index **order** is important. The index order is implementation-dependent, and array values may not be accessed in the order you expect. It is better to use a **for** loop, a **for of** loop, or **Array.forEach()** when the order is important.

Array forEach methods

To help with iteration, arrays also have a special method called **forEach**, it calls a function (a callback) once for each array element.

NOTE: **forEach() is slower than** a for loop

```
let x = [ "lightning", "apple", "squid", "speaker" ]
x.forEach(function(value, index, array) {
     console.log(`${value} is at index ${index}`)
})
```

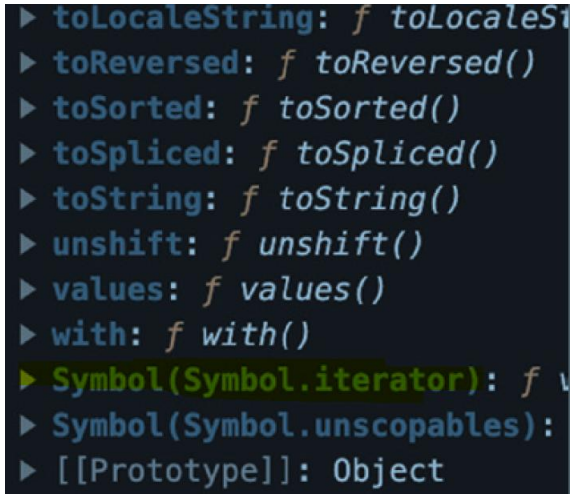
String Iteration - Since strings are also iterable, for...of and for...in also work on them.

```
> for(let item of "hello") {
      console.log(item)
  }
  h
  e
  l
  l
  o
```

Iteration Protocol

All types which have the **iteration protocol are iterable**.

You can see the iterator protocol by console logging an iterable's prototype, like console.log(**Array.prototype**)




```
let myArray = [ "lightning", "apple", "squid", "speaker" ]
let getIterator = myArray[Symbol.iterator]()
console.log(getIterator.next()) // {value: 'lightning', done: false}
```

6-code-readability.ts

Objects Are Not Iterable by Default

If you try to console log Object.prototype, you'll find **no Symbol, so it is not Iterable**. Fortunately, there is an easy way (below methods) to iterate through an object, and that is to convert it to an iterable data type, like an array.

- Object.keys() - which extracts all keys as an array
- Object.values() - which extracts all values as an array
- Object.entries() - which extracts all keys and arrays as an array of key–value pairs

```
let myObject = {
  firstName: "John",
  lastName: "Doe",
  age: 140
}
let myObjectKeys = Object.keys(myObject)
//Once we have extracted an array, then it is Iterable
for(let item of myObjectKeys) {
  console.log(item) //firstName, lastName, age
}
```

```
let myObjectVals = Object.values(myObject)
for(let item of myObjectVals) {
  console.log(item) //John, Doe, 140
}
```

```
let myObjectEntries = Object.entries(myObject)
for(const [key, value] of myObjectEntries) {//destruct
   console.log(`The key ${key} has a value ${value}`)
}
```

6-iterate-objects.ts

Keyed collections - Sets

Array limitations. e.g. can't have unique list, or can't have keys besides numbers, strings and symbols etc.

Sets (ES6 feature) will only support the addition of unique values for primitives, but possible to add two objects which have the same value (duplicate) which have different references.

```
let mySet = new Set()
mySet.add(4)
mySet.add(4)
console.log(mySet) // Set(1) {4}
```

```
let mySet = new Set()
mySet.add({ "name" : "John" })
mySet.add({ "name" : "John" })
console.log(mySet)
// Set(2) {{ "name" : "John" },
             { "name" : "John" }}
```

*Sets can be merged (**no duplicates**) using the three dots.*

```
let mySetOne = new Set()
mySetOne.add(4)
mySetOne.add(5)
let mySetTwo = new Set()
mySetTwo.add(5)
let bigSet = new Set([...mySetOne, ...mySetTwo])
console.log(bigSet) // Set{2} {4, 5}
```

Since sets are **iterable**, meaning for...of loops works fine, however, using for...in loops will not work since sets do not have indices.

```
for(let x of mySet) {
    console.log(x) // 4, 5, 6
}
for(let x in mySet) {
    console.log(x) // undefined
}
```

Finally, if the limitations on set methods (size, has, ..) or their lack of indices become too much, you can easily convert a set to an array:

```
let arrayFromSet = Array.from(mySet) // [ 4, 5, 6 ]
```

Set Keys and Values:

Just like objects, sets inherit keys(), values(), and entries() methods. On objects, the entries method returns arrays in the form **[key, value],** on Set it is **[value, value].** The three main methods for changing a set are: Set.add(),Set.delete(), Set.clear()

s-sets-maps.js

Array and Set compared

Traditionally **Arrays** are used to store elements in JavaScript, however, **Set** has some advantages:

- Performance - Deleting Array elements by value (arr.splice(arr.indexOf(val), 1)) is very slow. Checking existence of element in Set is fast.
- Set objects let you delete elements by their value. With an array, you would have to splice based on an element's index.
- The value NaN cannot be found with indexOf in an array.
- Set objects store **unique values**. You don't have to manually keep track of duplicates.

WeakSet object

WeakSet objects are collections of garbage-collectable values. The main differences to the Set object are:

- Compared to Sets, WeakSets are **collections of *objects or symbols only***, and not of arbitrary values of any type.
- The WeakSet is *weak (*references to objects in the collection are held weakly)
- WeakSets are not enumerable.
- The use cases of WeakSet objects are limited. They will not leak memory, so it can be safe to use DOM elements

Key and value equality of Map and Set

Both the key equality of Map objects and the value equality of Set objects are based on the SameValueZero algorithm:

- Equality works like the identity comparison operator === .
- -0 and +0 are considered equal.
- NaN is considered equal to itself (**contrary to** ===) .

```
function sameValueZero(x, y) {
  if (typeof x === "number" && typeof y === "number") {
    // x and y are equal (may be -0 and 0) or they are both NaN
    return x === y || (x !== x && y !== y);
  }
  return x === y;
}
```

NaN – NaN (Not-a-Number) is **not equal to any value (including itself, NaN !== NaN**) and is essentially an **illegal number value**, but *typeOf(NaN)===number //true*. Use isNaN(number) to check for NaNs.

Keyed collections - Maps

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length;      // person.length will return 0
var y = person[0];          // person[0] will return undefined
```

Maps (ES6 feature) are **iterable** by default, unlike objects. Just like sets, maps come with size(), add(), delete(), and clear() methods.

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
myMap.delete("key")
console.log(myMap) // Map(1) {'secondKey' => 'value'}
myMap.clear()
console.log(myMap) // Map(0) {}
```

Adding an entry to the Map **using square bracket notation** adds it to the object (same issue **like in Array**), at the same level as the prototype. It does not add an entry to the map itself, meaning you lose all the benefits of Maps

```
> let myMap = new Map()
  myMap.set("hello", "world")
  myMap[0] = "hello"

  console.log(myMap)

  Map(1) {'hello' => 'world'}
    [[Entries]]
      0: {"hello" => "world"}
      0: "hello"
      size: 1
    [[Prototype]]: Map
```

Maps allow keys to be of **any type** (objects only allow strings, numbers, or symbols). Also, a function key is possible in maps but not in objects.

```
let myMap3 = new Map()
myMap3.set(() => { return true }, "value")
console.log(myMap3)
// Map(1) { [Function (anonymous)] => 'value' }
```

Since maps can have keys of any type, retrieving non-primitive keys can become a little tricky. E.g. reference keys never garbage collected causes memory issues,… so use WeakMaps.

```
let myMap4 = new Map()
//we need to store the key in a variable to reliably retrieve it.
let someArray = [ 1, 2 ]
myMap4.set(someArray, "value")
const c = myMap4.get(someArray) // value
```

Merging, and Accessing Maps

```
let myMapOne = new Map()
myMapOne.set("key", "value")
let myMapTwo = new Map()
myMapTwo.set("secondKey", "value")
myMapTwo.set("key", "value2")
let bigMap = new Map([...myMapOne, ...myMapTwo])
console.log(bigMap) // Map(2) {'key' => 'value', 'secondKey' => 'value'}
console.log(bigMap.get('key')) //value2
```

Iteration methods: .keys(), .values(), .entries(), bigMap.forEach((value, key)

s-sets-maps.js

Object and Map compared

Traditionally **Objects** are used to map string to values in JavaScript, however, **Map** has some advantages:

- **Key Flexibility** - The keys of an Object are strings or symbols, whereas they **can be of any value** for a Map.
- **Size Determination** – Getting map size easy – size(), while you have to manually keep track of size for an Object or Object.keys(obj).length.
- **Iteration order** - The iteration of maps is in insertion order of the elements.
- For frequent additions and removals of key-value pairs, Map is more performant than using an Object.

Hints to decide whether to use a Map or an Object:

- Use maps over objects when keys are unknown until runtime, and when all keys are the same type and all values are the same type.
- Use maps if there is a need to store primitive values as keys because **object treats each key as a string** (no matter it's a num, bool, or etc)
- Use objects when there is logic that operates on individual elements.

WeakMap object

A WeakMap is a collection of key/value pairs (keys must be objects or non-registered symbols, values of any arbitrary JS type), which **does not create strong references to its keys**. Object's presence as a key in a WeakMap does not prevent the object from being **garbage collected**.

Console Errors, Debugs, Warnings, and Info

console,log(arguments) (browser vs Node.JS (FE vs BE))
In browser null, in Node.js (module, requires,..)

Console warnings, errors, … are accessed via console(.warn, .error, .info, .trace, ..clear())
*Console **assertions** let you show errors if certain criteria are found to be false*
*You can **group console logs** with console.group and console.groupCollapsed.*
***Console.table** gives us the ability to create table views inside the console*

```
console.assert(5 === 4, "the assertion is false")
console.log("Hi") // Top Level
console.groupCollapsed("Level 1") // Level 1 Group
console.log("Hi") // All of these are within Level 1 Group
console.log("Hi")
console.log("Hi")
console.group("Level 2") // Level 2 Group
console.log("Hi") // All of these are within Level 2 Group, which are within Level 1
console.log("Hi")
.. console.table(myFavoriteObject)
```

```
▼Assertion failed: the assertion is false
  (anonymous) @ r-browser-console.html:12
```

```
Hi
Level 1
  Hi
  Hi
  Hi
  Level 2
    Hi
    Hi
```

(index)	Values
name	'John'
age	105
place	'Planet Earth'

Console Timing/Counting

E.g. code that pauses execution via promises, waits for APIs to load, or has inefficient calc. that result in long wait times. All of these impact page load time

Therefore, optimizing your JavaScript is really important, and the best way to find code that is performing code is **by measuring how long it takes to run**.
To help diagnose these problems, there are three console methods we can use.

- console.time
- console.timeLog
- console.timeEnd
- console.count
- console.countReset

Errors and Exceptions – Error Handling in JavaScript

Exceptions are an amazing mechanism **to favor clean code by separating concerns** - happy scenario from failed case and dealing with the latter elegantly.

JavaScript/Typescript is quite good at resolving most coding mistakes, so errors really are the last resort (beyond our control)!

- We should always take steps to mitigate the error, or report it back to the user
- Users/customers don't like data loss!

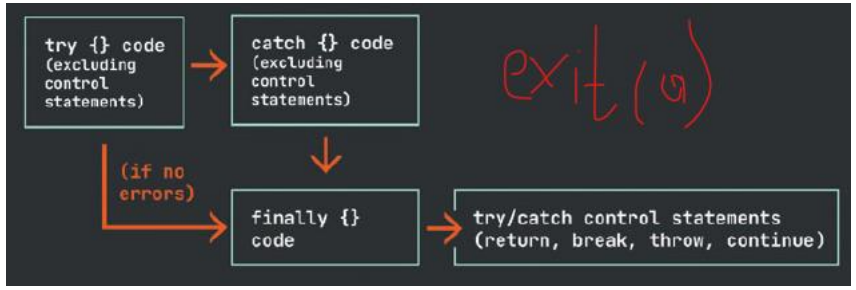
Throw, and Try...Catch...Finally

```
let num = 1, x = 5;
try {//code block to run
  num.toUpperCase();   //cannot convert to upper case
  x = y + 1;   // y cannot be used (referenced)
  let jsonData = JSON.parse(inputData)//non-json input
} catch(err) {//code block to handle errors
  document.getElementById("demo").innerHTML = err.name;
  console.error(`${e.name}: ${e.message}`);
} finally() {
  //runs always, does clean up or release resources
  let JSONData = {}
  console.log(JSONData)
}
```

- cause (optional)
- message
- name
- stack (optional)

Only standard field Error object has is the **message** property. Others platform dependent.

Error Name (standalone object) - Categories	Description
EvalError	An error has occurred in the eval() function
RangeError	"out of range" once you try access smt. out of size
ReferenceError	An illegal reference has occurred, non-existing one
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred
AggregateError	An error which contains many errors.
InternalError	An error occurred inside the JavaScript engine



The throw Statement

The `throw` statement allows you to create/generate a custom error.
Technically you can **throw an exception (throw an error)**.
The exception can be a JavaScript `String`, a `Number`, a `Boolean` or an `Object`:

```
throw "Too big";     // throw a text
throw 500;           // throw a number
throw new TypeError("Type Error") // throw a TypeError
throw new Error("A Provisioning Error"); // throw an Error
```

Always throw an actual error object (not CODE) to preserve information about the error

Error Handling Best Practices

In Java 'I' (pipe is used for multiple exceptions)

Sometimes, a part of your code can be known to have the potential to throw **multiple types of errors**.

Use Try / Catch Correctly

- Never return a string or CODE instead of the error
- Never "swallow or ignore" errors, always do something with them
- Many linters warn you about empty exception blocks.
- Explicitly type the error as **`unknown`** for readability
- In a try / catch we should always narrow the error down to a concrete/specific type
- Don't Return NULL or UNDEFINED
- Don't nest exceptions - see sample
- Use Exceptions(IDs & codes) rather than Return Codes – pollute with happy scenario
- Throw Early and Catch Late

Handling a specific error type

```
try {
    JSON.parse("OK")
    } catch(e) {
    if(e instanceof SyntaxError) {
       console.error("This is a syntax error");
    } else if(e instanceof EvalError) {
       console.error("This was an error with the eval() function");
    } else if(e instanceof MyAppError) { //my error, extends Error
       console.error("Application error: "+e);
    }
}
```

```
files.forEach(file => {
  try {
    const filePath = path.resolve(process.env.HOME, file);
    const data = fs.readFileSync(filePath);
    console.log('File data is', data);
  } catch (err) {
    if (err.code === 'ENOENT') {
      console.error('File not found');
    } else {
      throw err;
    }
  }
});
```

code (DOMException) - Deprecated: This feature is no longer recommended. Though some browsers might still support it, it may have already been removed from the relevant web standards
BUT in Node.js deal with 'CODE', e.g. ENOTDIR, etc

Benefits of Custom Error Classes

Better identify the errors the application needs to handle
Work with errors in a type-safe way
Work with different errors in a consistent way
Enhance errors with custom functionality

Give Useful Errors & Error Documentation

- Provide detailed information about each error including:
- What caused the error to be thrown
- Useful name, description, error code
- Whether the error is thrown or returned
- **Use JSDoc's @throw tag to provide information on thrown errors**
- Document all Errors - useful for developers
- Consider errors that need to be displayed (Hiding Low-Level Errors) to the user

```
/**
 * @throws {StorageError} When LocalStorage is full or disabled
 */
```

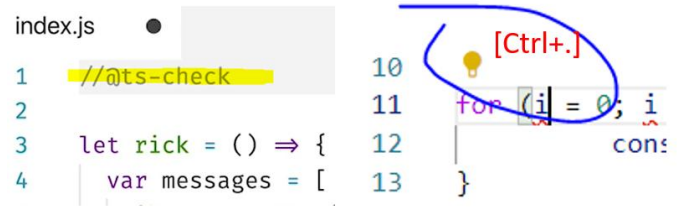
z--error.js

Throwing vs. Returning

Throw when the error cannot be recovered from, return when it can
Use a union type including `Error` when a function returns an error

Debugging Experience

VSCode default TS Compiler: [**@ts-check**]: or per-prj implicitProjectConfig.**checkJS**: true

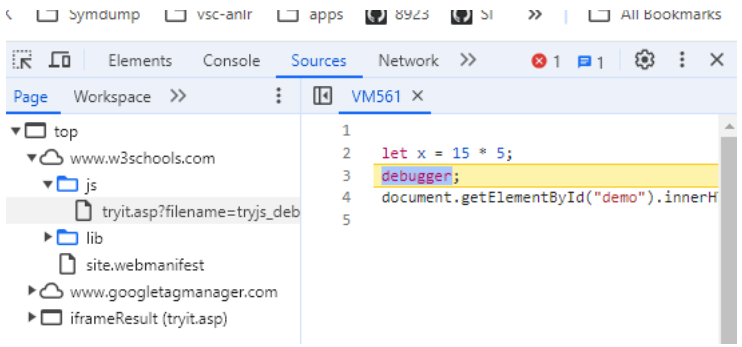


Using **console.error()** rather than **console.log()** is advised for debugging
If your browser supports debugging, use console.log() to display JavaScript values in the debugger window

The debugger Keyword

The debugger keyword stops the execution of JavaScript, and calls (if available) the debugging function.
This has the same function as setting a breakpoint in the debugger.
If no debugging is available, the debugger statement has no effect.
With the debugger turned on, this code will stop executing before it executes the third line.

```
let x = 15 * 5;
debugger;
document.getElementById("demo").innerHTML = x;
```



Setting Breakpoints

In the debugger window, you can set breakpoints in the JavaScript code.

Major Browsers' Debugging Tools

Normally, you activate debugging in your browser with F12, and select "Console" in the debugger menu.

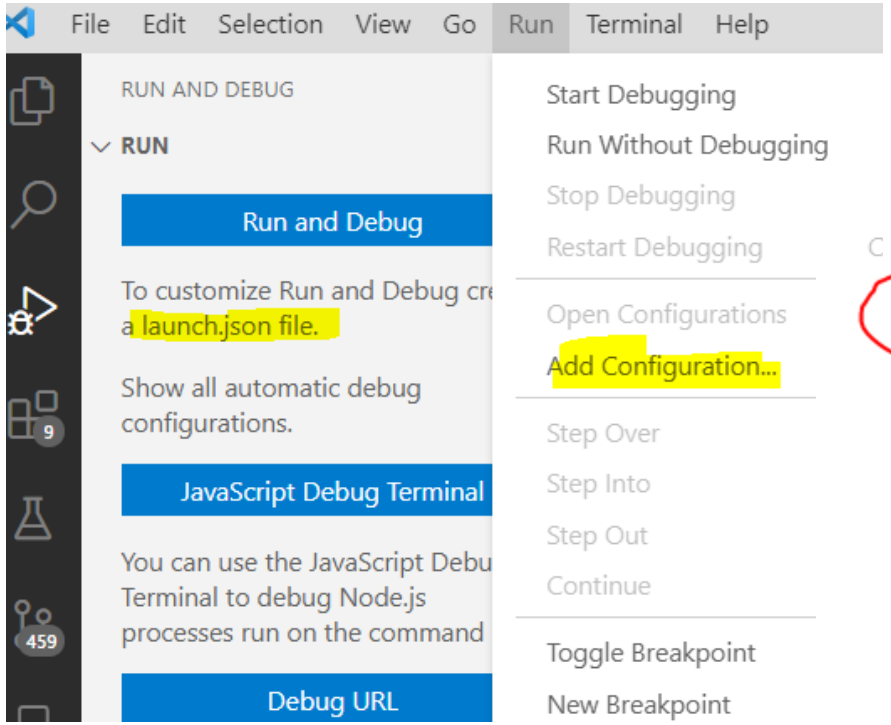
z-js-check-with-ts.js

[Ctrl+.]

Debugging Experience

Debugger Extensions

- **Built-in** debugging support for the Node.js runtime and can debug JS, TS, or any other that gets transpiled to JS
- To debug other languages and runtimes either search extensions or from menu **Run -> Install Additional Debuggers**

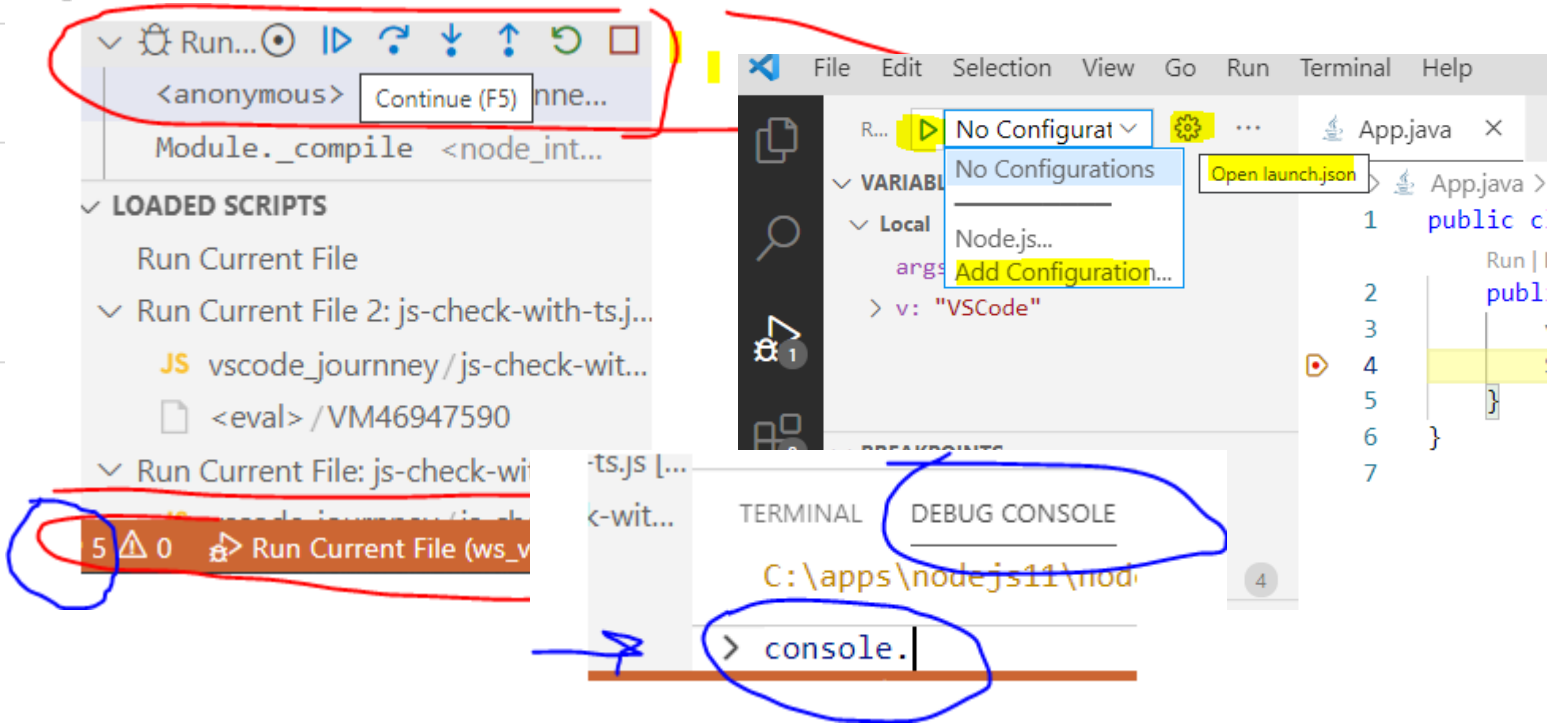


Launch Configuration (compound, .. DevTool-F12, ..)

Create “launch.json” either by ‘gear-icon’ or Run->Add Conf.

E.g. - Add "**stopOnEntry**": true

- Drag&drop for Execution point

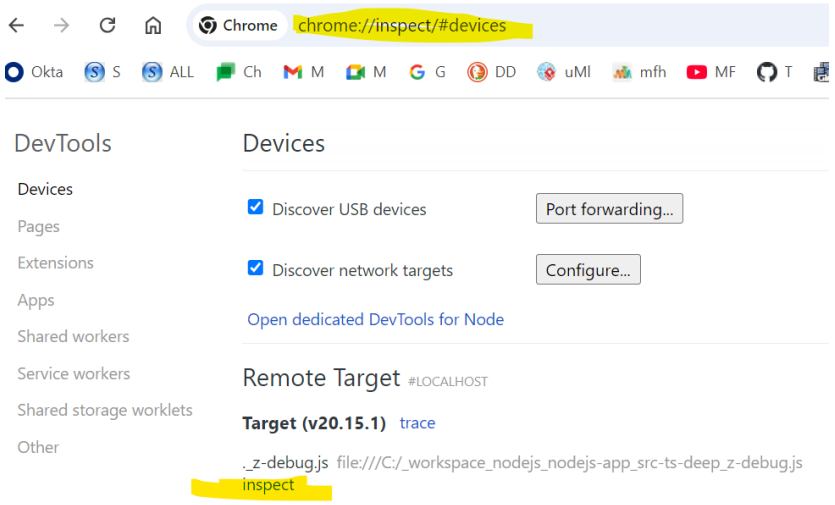


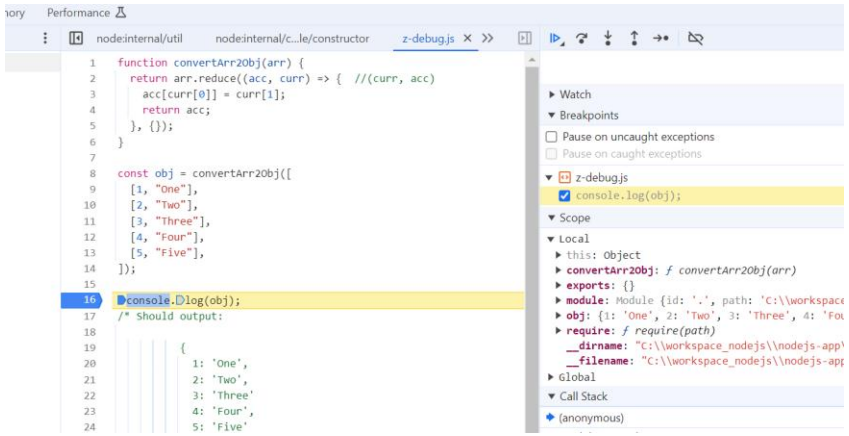
Debugging Experience

Using Chrome Inspect Tools

- ❑ Debugger (Node built-in) and more utilities
- ❑ **Debug with Chrome dev tools**
- ❑ **You can use it on (Nodje.js process attach)**

```
>node .\z-debug.js
>node --inspect-brk .\z-debug.js
>chrome://inspect    (in Chrome)  Click “inspect”  and Ctrl+P
```





What are source maps?

Crucial tool in modern web development that make debugging significantly easier.

sourceMaps – maps transpiled JS code to TS. All browsers support this. Helpful for debugging .

Source map files contain essential information about how the compiled code maps to the original code.

```
{
  "mappings": "AAAAA,SAASC,cAAc,WAAWC, ...",
  "sources": ["src/script.ts"],
  "sourcesContent": ["document.querySelector('button')..."],
  "names": ["document","querySelector", ...],
  "version": 3,
  "file": "example.min.js.map"
}
```

Code Beatify Experience

Linters and formatters are not the same thing, although there can be overlap

Formatters are concerned with the presentation of code in files

We can configure whether semi-colons are added at line ends, tabs vs spaces, etc.

Automation options

- Editor integration
- Run Prettier on every file save
- Show lint warnings as you code
- Git hooks
- Pre-commit

The Purpose of Formatting

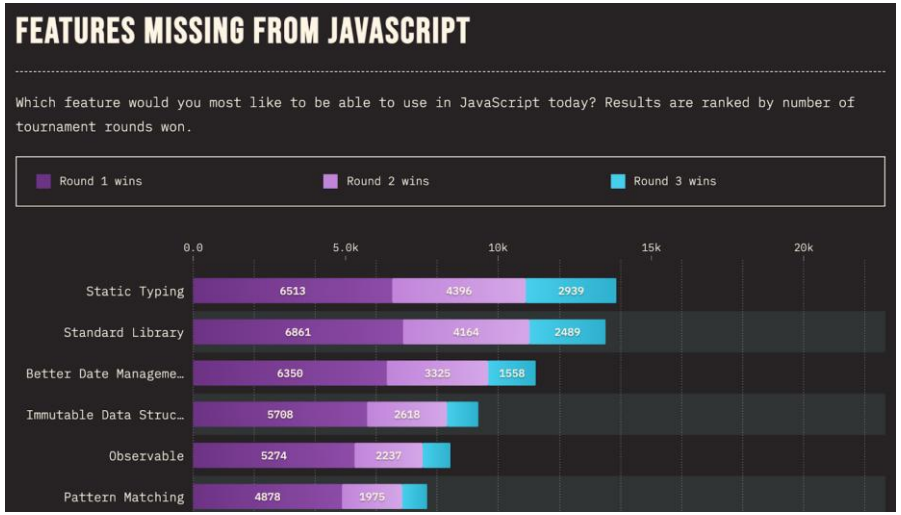
It is about communication, and communication is the professional developer's **first order of business(?)**.
Todays functionality can be changed tomorrow - but the readability of code will have a profound effect on all changes

- **Vertical formatting/ordering** – How big file size should be? Small files are usually easier to understand than large files are
- **The Newspaper Metaphor** – You read it vertically (header, synopsis, details later,). Source file to be like a newspaper article
- **Vertical Density** - So lines of code that are tightly related should appear vertically dense.
- **Vertical Distance** - Closely related concepts should not be separated into different files unless you have a very good reason.
- **Variable** (Local, Instance, Static) **Declarations, Dependent Functions** – should be declared as close to their usage as possible.
- **Conceptual Affinity** – Certain bits of code want to be near each other – less vertical distance between (group of functions calling each other etc..)
- **Horizontal Formatting** – How wide should a line be? 80-120 chars ideal? Scrolling right is worse than scrolling down.
- **Horizontal Density** – How using horizontal white spaces (some tools - reformatting code are blind to the precedence of operators)
- **Horizontal Alignment** – manual one not useful
- **Indentation** - Without indentation, programs would be virtually unreadable by humans. Not mix up Python with others.
- **Team Rules** – Team Decision e.g. Uncle Bob's Formatting Rule

Modern Javascript

Node.js != JS

- ES TC39, https://github.com/tc39, TC-39 Process, Ecma-262
- V8 Engine will follow implementing TC39
- Yearly Releases since ES2015 [ES6], ES2016, ….
- 5 StagedProcess [0-Strawman,1-Proposal,2-Draft,3-Candidate,4-Finished]
- Babel - faster
- Variables and Block Scopes, Object Literals
- Arrow Functions
- Destructing and Rest/Spread
- Inheritance – state and action based (in Java action based)
- Promises and Async/Await
- Classes,…
- **Templates-string (interpolation, ` ${dyno-exp}`)** (also like **multi-lines** Java 13)
- **Dynamic properties**
- **Not yet – TS (type, type aliases, etc…)**
- **BUT, Good bye TS time is approaching –Native JS Proposals (Type Annotation) coming**



Prefer ECMAScript Features to TypeScript Features

JavaScript ES5

ECMAScript 2009, also known as ES5, was the **first major** revision to JavaScript. See ES features ..

`"use strict"` defines that the JavaScript code should be executed in "strict mode".

The `charAt()` method returns the character at a specified index (position) in a string:

ES5 lets you define object methods with a syntax that looks like getting or setting a property.

With the `bind()` method, an object can borrow a method from another object.

ES5 Features

- "use strict"
- String[*number*] access
- Multiline strings
- String.trim()
- Array.isArray()
- Array forEach()
- Array map()
- Array filter()
- Array reduce()
- Array reduceRight()
- Array every()
- Array some()
- Array indexOf()
- Array lastIndexOf()
- JSON.parse()
- JSON.stringify()
- Date.now()
- Date toISOString()
- Date toJSON()
- Property getters and setters
- Reserved words as property names
- Object.create()
- Object.keys()
- Object management
- Object protection
- Object defineProperty()
- Function bind()
- Trailing commas

JavaScript ES6

ECMAScript 2015 a.k.a ES6 **second major** revision of JS

The `let` keyword allows you to declare a variable with block scope.

The JavaScript `for/of` statement loops through the values of an iterable objects.

New Features in ES6

- ✓ The let keyword
- ✓ The const keyword
- ✓ Arrow Functions
- ✓ The {a,b} = Operator
- ✓ The [a,b] = Operator
- ✓ The ... Operator
- ✓ For/of
- ✓ Map Objects
- ✓ Set Objects
- ✓ Classes
- ✓ Promises
- ✓ Symbol
- ✓ Default Parameters
- ✓ Function Rest Parameter
- ✓ String.includes()
- ✓ String.startsWith()
- ✓ String.endsWith()
- ✓ Array entries()
- ✓ Array.from()
- ✓ Array keys()
- ✓ Array find()
- ✓ Array findIndex()
- ✓ Math.trunc
- ✓ Math.sign
- ✓ Math.cbrt
- ✓ Math.log2
- ✓ Math.log10
- ✓ Number.EPSILON
- ✓ Number.MIN_SAFE_INTEGER
- ✓ Number.MAX_SAFE_INTEGER
- ✓ Number.isInteger()
- ✓ Number.isSafeInteger()
- ✓ New Global Methods
- ✓ JavaScript Modules

ECMAScript 2016

Old ECMAScript versions was named by numbers: ES5 and ES6.
From 2016, versions are named by year: ES2016, 2018, 2020 ...

New features in ECMAScript 2016:

- JavaScript Exponentiation (**): `let x = 5; let z = x ** 2; //25`
  `x ** y` produces the same result as `Math.pow(x, y)`
- JavaScript Exponentiation assignment (**=) `let x = 5; let z = x ** 2; //25`
- JavaScript Array includes() - if an element is present in an array

Prefer ECMAScript Features to TypeScript Features

ECMAScript 2017

New features in ECMAScript 2017:

- JavaScript String padding - `padStart()` and `padEnd()` to support padding at the beginning and at the end of a string
- JavaScript Object entries() - `Object.entries()` returns an array of the key/value pairs in an object
- JavaScript Object values() - `Object.values()` is similar to `Object.entries()`, but returns a single dimension array of the object values

- JavaScript async and await
- Trailing Commas in Functions

JavaScript allows trailing commas wherever a comma-separated list of values is accepted. In Array and Object Literals, Function Calls, Parameters, Imports and Exports. Not good for JSON, …

```
function myFunc(x,,,) {};
const myArr = [1,2,3,4,,,];
const myObj = {fname: John, age:50,,,};
```

- JavaScript Object.getOwnPropertyDescriptors

ECMAScript 2018

New Features in ECMAScript 2018

- Asynchronous Iteration - With asynchronous iterables, we can use the `await` keyword in `for/of` loops. E.g. `for await () {}`
- Promise Finally - finalizes the full implementation of the Promise object with Promise.finally
- Object Rest Properties - allows us to destruct an object and collect the leftovers onto a new object
- New RegExp Features - added 4 new RegExp features, (\p{...}), (?<=) and (?<!), s (dotAll) Flag
- JavaScript Shared Memory

```
let myPromise = new Promise();

myPromise.then();
myPromise.catch();
myPromise.finally();
```

JavaScript Threads - use the Web Workers API (for Browser based app) or Worker Threads (for NodeJS) to create threads

JavaScript Shared Memory: Shared memory is a feature that allows threads to access and update the same data in the same memory.
Instead of passing data between threads, you can pass a SharedArrayBuffer object that points to the memory where data is saved.
A **SharedArrayBuffer** object represents a fixed-length raw binary data buffer similar to the ArrayBuffer object.

ECMAScript 2019

- String.trimStart() - method works like trim(), but removes whitespace only from the start of a string.
- String.trimEnd() - works like trim(), but removes whitespace only from the end of a string.
- Object.fromEntries - method creates an object from iterable key / value pairs
- Optional catch binding - you can omit the catch parameter if you don't need it
- Array.flat() - creates a new array by flattening a nested array
- Array.flatMap() - first maps all elements of an array and then creates a new array by flattening the array
- Revised Array.Sort()
- Revised JSON.stringify()
- Separator symbols allowed in string litterals
- Revised Function.toString()

```
const myArr = [[1,2],[3,4],[5,6]];

const newArr = myArr.flat();//1,2,3,4,5,6
```

```
const myArr = [1, 2, 3, 4, 5,6];
const newArr = myArr.flatMap(x => [x, x * 10]);//1,10,2,20,3,30,4,40,5,50,6,60
```

New Features in ES2020

- BigInt - used to store big integer values that are too big to be represented by a a normal JavaScript Number
- String matchAll()
- The Nullish Coalescing Operator (??) - returns the first argument if it is not **nullish** (null or undefined).
- The Optional Chaining Operator (?.) – e.g. let name = car?.name;
- Logical AND Assignment Operator (&&=)
- Logical OR Assignment (||=)
- Nullish Coalescing Assignment (??=)
- Promise.allSettled() - method returns a single Promise from a list of promises

```
let name = null;
let text = "missing";
let result = name ?? text;
//like Elvis operator in Spring (?:)
```

New Features in ES2021

- Promise.any()
- String replaceAll()
- Numeric Separators (_) – e.g. const num = 1_000_000_000;

New Features in ES2022

- ✓ Array at()
- ✓ String at()
- ✓ RegExp /d
- ✓ Object.hasOwn()
- ✓ error.cause
- ✓ await import
- ✓ Class field declarations
- ✓ Private methods and fields

New Features in ES2023

- Array findLast()
- Array findLastIndex()
- Array toReversed()
- Array toSorted()
- Array toSpliced()
- Array with()
- #! (Shebang) – like Java 11 feature

New Features in ES2024

- Object.groupBy() //like Java stream API
- Map.groupBy()
- Temporal.PlainDate() – only date
- Temporal.PlainTime() – only time
- Temporal.PlainMonthDay() – no year
- Temporal.PlainYearMonth()

JS Reference,

Demo available: Testing Frameworks and Tools, Code Coverage

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