Introduction to JavaScript, Json, TypeScript, TSX

Cours Esilv 2024

arnaud.nauwynck@gmail.com

Outline

JavaScript

JSON

TypeScript

TSX

JavaScript

Java Script

Syntax inspired from "Java"
(itself inspired from "C", "C++")

untyped at compile-time,
interpreted at runtime
like "script" shells.

only "string", "numeric", boolean".
no "class" or type declarations !

Birth of JavaScript : 1995

ChatGPT

history of javascript

The History of JavaScript

1. Birth of JavaScript (1995)

JavaScript was created in 1995 by Brendan Eich while he was working at Netscape Communications Corporation. Originally, the language was developed in just 10 days and was initially named Mocha, then renamed to LiveScript, and finally to JavaScript.

Key Points:

- **Creator:** Brendan Eich
- **Initial Name:** Mocha
- **Release Date:** December 1995
- **Initial Purpose:** To add interactivity to web pages

JavaScript Evolution

Birth of JS

used in Netscape web browser

Adopted by browsers, but not standard

submitted to ECMA international for standardisation

spec : "**ECMAScript**"

AJAX and Web 2.0

standardisation of browsers

ECMAScript 5
strict mode, JSON, ...

birth of many frameworks (jQuery, AngularJS, etc..)

birth of **Node.JS**
server-side JS development ecosystem (npm, grunt, gulp, ..)

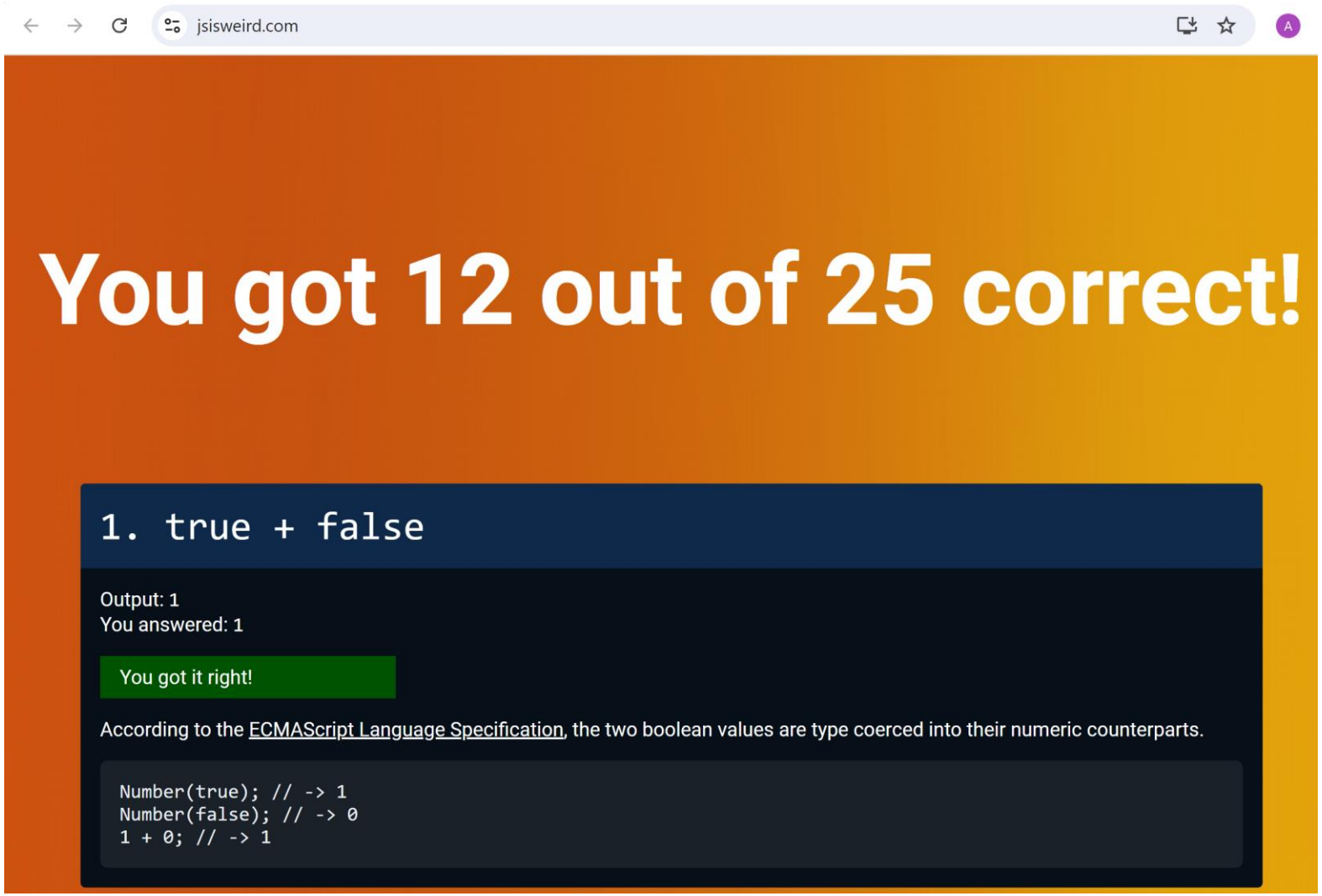
TypeScript

ECMAScript 6 (=2015)
class, arrow, modules, ...

developed in 10 days by 1 person
... => explain many weird parts of the langage

https://jsisweird.com/

even for me



There are (strange) implicit coercions between primitive types...
For exact comparison, use "===" ("==" may introduce conversions)

JavaScript = un-maintanable code !!

NO Type !!

.. alternative like Google Dart since 2011 (ex in Flutter)

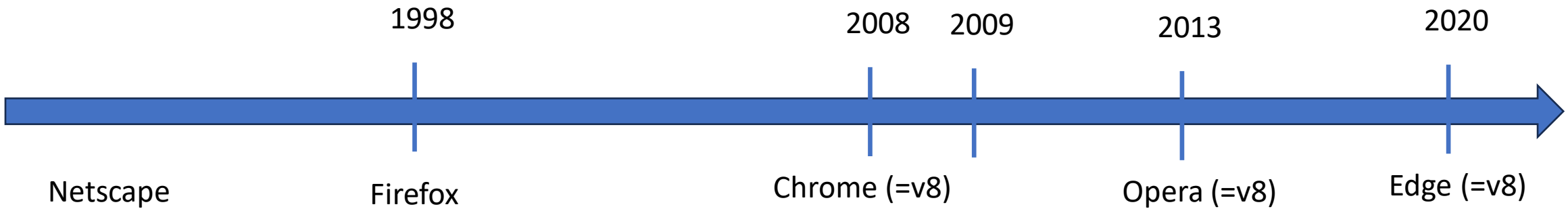
.. work-around since 2012 via TypeScript

NO Class !

.. work-around since 2012 via TypeScript

.. ok since 2015 EcmaScript6

Runtime Implementations



Mozilla
Open-Source
organisation
created from
Netscape Company

Firefox from
Nescape Navigator

Internet Explorer

IE had is own runtime.
his own non-standard specs
=> lot of behavior differences

jQuery was a hack for compatibility

Node.JS re-use the same runtime
as Chrome : "V8"

many browsers followed
(Opera in 2013, Edge in 2020, ...)

JavaScript Primitive Value Objects

null

undefined

String

Numeric

Boolean

Nested Compositions: [Array], {Object}

Array

```
> [ undefined, null, true, 12.3, "text",  [], {}, [{}, {}] ]
< ▶ (8) [undefined, null, true, 12.3, 'text', Array(0), {…}, Array(2)]
```

Object

```
> { a: undefined, b: null, c: 12.3, d: "text", e: [], f: {}, g: [{}, {}] }
< ▶ {a: undefined, b: null, c: 12.3, d: 'text', e: Array(0), …}
```

JSON = JavaScript Object Notation

NO code, only Value

Can be integrated directly in JavaScript (or TypeScript)

=> SUPER Easy for Http Rest using JSON as data encoding protocol

in pure JSON, identifier need to have double-quotes
... NOT needed in Js/Ts

JSON Example

```
{
  "a": null,
  "b": undefined,
  "c": 123,
  "d":  true,
  "e":  "text with escaped \" and \n chars",
  "f":  [  null, undefined, 123, true, [], {} ],
  "g": {  "subField": 1, "b": true }
}
```

Example Unquoted JSON directly in JavaScript

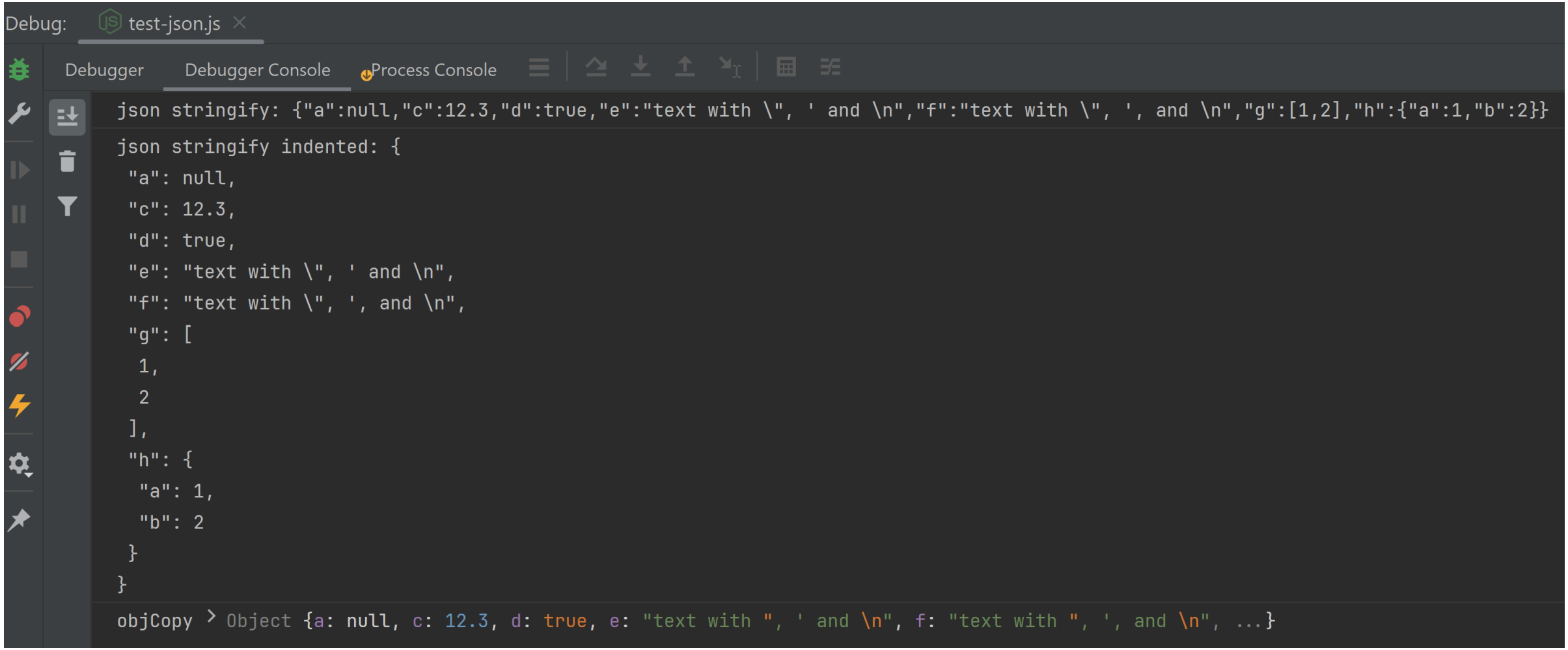
```
let obj = {
   a: null,
   b: undefined,
   c: 123,
   d:  true,
   e:  "text with escaped \" and \n chars",
   f:  [  null, undefined, 123, true, [], {} ],
   g: {  subField: 1, b: true }
};
```

JSON stringify() / parse()... builtin in JavaScript

test-json.js

```
let obj : {...} = { a:null, b:undefined, c:12.3, d:true,
    e: 'text with ", \' and \n',
    f:"text with \", ' and \n",
    g: [1, 2],
    h: { a:1, b:2}
}
let objJson : string = JSON.stringify(obj);
console.log('json stringify:', objJson);

let objJsonIndented : string = JSON.stringify(obj, replacer: null, space: ' ');
console.log('json stringify indented:', objJsonIndented);

let objCopy = JSON.parse(objJson);
console.log('objCopy', objCopy)
```

JSON.stringify(obj), .parse(text)



JavaScript object deep clone

```
let objCopy = JSON.parse(JSON.stringify(obj));
```

Objects (~= LinkedHashMap<String,*>)

```
> let  obj = { }

  obj.x = 1

  obj.x = "changed"

  console.log(obj)
  ▶ {x: 'changed'}
< undefined
> obj
< ▶ {x: 'changed'}
```

using field Array notation :

```
> obj['x']
< 'changed'
> obj['x'] = true
< true
```

Creating Object with Field Values

Equivalent Code in Java

```
> a = { x:1, y:2}
< ▶ {x: 1, y: 2}
```

```
a = new LinkedHashMap<String,Object>();
a.put('x', 1);
a.put('y', 2);
```

```
> let x = 1, y = 2;
< undefined
> a = { x, y }
< ▶ {x: 1, y: 2}
> b = { x:1, y }
< ▶ {x: 1, y: 2}
```

avoid stamming { "x": x, "y": y }
or { x: x, y: y }

Example with Structured output from console.log()

```
9   let obj :{…}  = { a,b,c,d,e,f,g};
10
11  console.log('message with structured', obj)
12
13  console.log('message with a,b', { a, b})
14
15  console.log('message with [a,b]', [ a, b ])
```

Object Spreading Operator

```
> obj1 = { x:1, y: 2 }
```

```
> obj2 = { ...obj1, z: 3 }
◂· ▶ {x: 1, y: 2, z: 3}
```

add all properties from obj1,
then complete with values

```
> obj3 = { ...obj1, y:4 }
◂· ▶ {x: 1, y: 4}
```

... then override with values

```
> obj4 = { y:4, ...obj1 }
◂· ▶ {y: 2, x: 1}
```

... order matters
(first define order, last override value)

```
> objShallowCopy = { ...obj1 }
◂· ▶ {x: 1, y: 2}
```

Undefined Field, delete obj.field

```
> let a= { x: 1 }
< undefined
> a.y
< undefined
> a.y = 2
< 2
> a
< ▶ {x: 1, y: 2}
> delete a.x
< true
> a
< ▶ {y: 2}
```

get unknown field
=> NO error, return undefined

add new field on first set

delete field

set field undefined

```
> a.y = undefined
< undefined
> a
< ▶ {y: undefined}
> a.y
< undefined
```

Object De-Structuring to Variables

```
> obj = { x:1, y:2, z:3 }
```

```
> let {x} = obj
< undefined
> x
< 1
```

extract property "x" from obj, assign to local variable x

```
> let {z,y} = obj
< undefined
> y
< 2
```

can extract many properties partially
order does not matter, only names

Object De-Structuring to Function Parameter

```
> function f({a, b})  { console.log('call f', { a, b }); }
< undefined
```

```
> f({a:1, b: 2})
call f ▶ {a: 1, b: 2}
```

```
> obj = { a:1, b: 2}
< ▶ {a: 1, b: 2}
> f(obj)
call f ▶ {a: 1, b: 2}
```

Function Param De-Structuring ... undefined

ANNEXE

```
> f({a:1})
call f ▶ {a: 1, b: undefined}
```

object param is defined, partially filled

```
> f({})
call f ▶ {a: undefined, b: undefined}
```

object param is defined, but empty

```
> f()
✕ ▶Uncaught TypeError: Cannot destructure property 'a' of 'undefined' as it is
undefined.
    at f (<anonymous>:1:13)
    at <anonymous>:1:1
```

object param is undefined !

```
> let {a} = undefined
✕ ▶Uncaught TypeError: Cannot destructure property 'a' of 'undefined' as it is
```

Iterating on Fields, field order, owned field

ANNEXE

Equivalent Code in Java

```
> Object.getOwnPropertyNames(a)
<· ▶ (2) ['x', 'y']
```

a.keySet()

```
> for(let e of Object.entries(a)) {
      console.log(e);
  }
  ▶ (2) ['x', 1]
  ▶ (2) ['y', 2]
```

a.entries()

```
> a = { y:1, x:2 }
<· ▶ {y: 1, x: 2}
```

notice order: y,x instead of x,y

a = new LinkedHashMap<String,Object>()
... (!= HashMap)

Notice, can use real "new Map" in JavaScript

ANNEXE

```
> m = new Map()
<· ▶ Map(0) {size: 0}
> m.set('key1', 1)
<· ▶ Map(1) {'key1' => 1}
> m.get('key1')
<· 1
> m
<· ▶ Map(1) {'key1' => 1}
```

Array

Equivalent Code in Java

```
arr = [1, 2, 3, 4, 5]
```

```
arr = new ArrayList<>(
        Arrays.asList(1, 2, 3, 4, 5));
```

```
> arr.push(6)
    > arr
    ◂ ▶ (6) [1, 2, 3, 4, 5, 6]
```

```
a.add(6);
```

```
> arr.splice(3, 1)
    > arr
    ◂ ▶ (5) [1, 2, 3, 5, 6]
```

```
a.removeAt(3);
```

Array operations

```
arr = [1, 2, 3, 4, 5]
```

Equivalent Code in Java

```
> arr.splice(3, 0, 1000)
```

arr.insertAt(3, 1000);

```
> arr
< ▶ (6) [1, 2, 3, 1000, 4, 5]
```

```
> arr.indexOf(1000)
< 3
```

arr.indexOf(1000);

```
> arr.findIndex( x => x === 1000)
< 3
```

no simple equivalent ?

```
> arr.splice(arr.indexOf(1000), 1)
```

arr.remove(1000)

Array Functional Operations

```
arr = [1, 2, 3, 4, 5]
```

Equivalent Code in Java

```
> arr.filter(x => x > 3)
< ▶ (2) [4, 5]
```

```
b = arr.stream()
    .filter(x -> x > 3)
    .collect(Collector.toList())
```

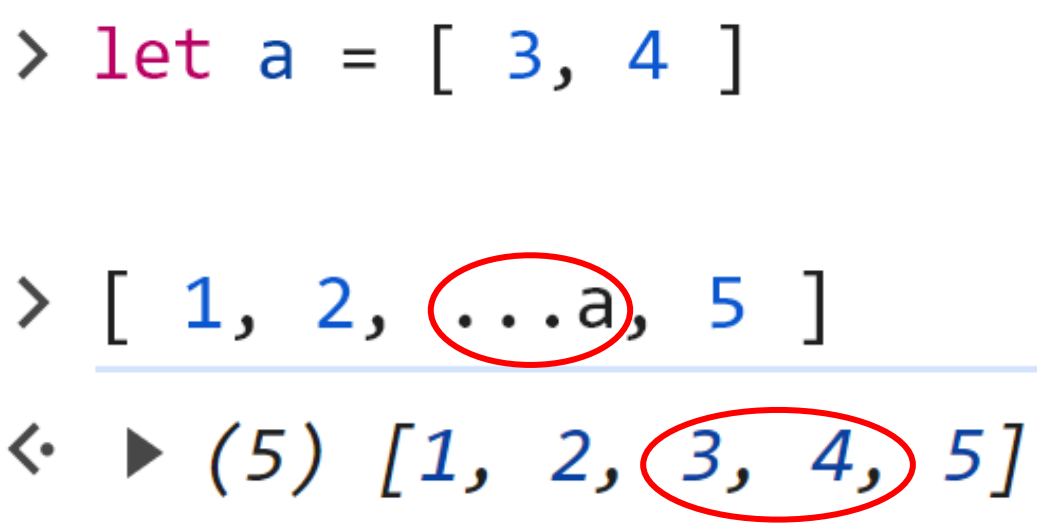
```
> b = arr.map(x => x*2)
< ▶ (5) [2, 4, 6, 8, 10]
```

```
b = arr.stream()
    .map(x -> x*2)
    .collect(Collector.toList())
```

```
> sum = arr.reduce((a,b)=>a+b, 0)
< 15
```

```
b = arr.stream()
    .reduce(0,  (a,b) -> a+b)
```

Array Spreading Operator



Array Destructuring to Variables

```
> arr = [1, 2, 3, 4, 5]
< ▶ (5) [1, 2, 3, 4, 5]
```

```
> [a, b, ...c] = arr
< ▶ (5) [1, 2, 3, 4, 5]
> { a, b, c }
< ▼ {a: 1, b: 2, c: Array(3)} i
    a: 1
    b: 2
  ▶ c: (3) [3, 4, 5]
  ▶ [[Prototype]]: Object
```

```
> [a, b] = arr
< ▶ (5) [1, 2, 3, 4, 5]
> {a, b}
< ▶ {a: 1, b: 2}
```

Functions

```
> let f = function(a, b) { console.log('f()', {a,b}); }
< undefined
> f(1)
  f() ▶ {a: 1, b: undefined}
< undefined
> f(1, 2, 3)
  f() ▶ {a: 1, b: 2}
< undefined
```

Function default parameter Value

```
> f = function(a, b=3) { return a+b; }
```

```
> f(1)
<· 4
```

call function without parameter "b" => using default

```
> f(1,1)
<· 2
```

call function with explicit parameter override

Idiomatic ("Strange") JS Code using default parameter options object

```
> f = function (a, opt={ b:1, c:undefined }) {
      let tmpB = opt?.b || 1;     // default to 1
      let tmpC = opt?.c || tmpB; // default to b if unset
      return a + tmpB + tmpC;
    }

> f(1)
< 3
> f(1, {b:2})
< 5
> f(1, {c:2})
< 4
> f(1, {})
< 3
```

Functions are Objects

```
> f = function(a,b) { console.log('f', {a,b}); }
< ƒ (a,b) { console.log('f', {a,b}); }
```

```
> obj = {...f }
< ▶ {}
```

object spreading on "function object" ! => empty by default

```
> f.x = 1
< 1
```

can assign field to "function object"

```
> f
< ƒ (a,b) { console.log('f', {a,b}); }
```

displaying "function object", still only show source code (no fields)

```
> f.x
< 1
```

can get field from "function object"

```
> obj = {...f }
< ▶ {x: 1}
```

object spreading ... all like object

Function <-> String source code

```
> fSourceCode = f.toString()
< "function(a,b) { console.log('f', {a,b}); }"

> eval("1 + 1")
< 2
> eval(" g = " + fSourceCode)
< ƒ (a,b) { console.log('f', {a,b}); }
> g
< ƒ (a,b) { console.log('f', {a,b}); }
> f === g
< false
```

Arrow (Lambda) Function

```
> let myFunc = (a,b) => { console.log('myFunc()', {a,b}); }
< undefined
> myFunc(1, 2)
  myFunc() ▶ {a: 1, b: 2}
< undefined
> let myFuncPlus = (a,b) => a+b
< undefined
> myFuncPlus(1, 2)
< 3
```

arrow function with code block

arrow function with expression

Function used as field ~ method ? this = ??

```
> let myFuncThisX = function() { return this.x; }   // strange... see next
< undefined
> let a = { x:1, getX: myFuncThisX }
< undefined
> a.getX()
< 1
```

"this" bounded to function

```
> let axFnc = getX.bind(a)
< undefined
> axFnc()
< 1
```

Function as "Class" Constructor

```
> let MyConstructor = function(x) { this.x = x; this.y = 0; }
< undefined
> let pt = new MyConstructor(12);
< undefined
> pt
< ▶ MyConstructor {x: 12, y: 0}
```

prototype

adding field (property / method) to object prototype => adding to all impacted instances

```
> MyConstructor.prototype.getX = function() { return this.x; }
< ƒ () { return this.x; }
> MyConstructor.prototype.getY = function() { return this.y; }
< ƒ () { return this.y; }
> pt.getX()
< 12
> pt.getY()
< 0

> MyConstructor.prototype.t = 123
< 123
> pt.t
< 123
```

emulated class with Prototype (before EcmaScript2015)

using Typescript (transpiler to JavaScript)

class are emulated with prototype before EcmaScript2015

looks like class after (but internally, still mutable prototype obects)

Class in JavaScript !

```
> class A { x; y }
< undefined
```

define a class

```
> A
< class A { x; y }
```

```
> A.toString()
< 'class A { x; y }'
```

display class source

```
> a = new A()
< ▶ A {x: undefined, y: undefined}
```

new object
(no constructor yet, default)

Class constructor

```
> class A {
      constructor(x) { this.x = x; }
  }
```

```
> a = new A(1);
< ▶ A {x: 1}
```

new call constructor

```
> a = new A();
< ▶ A {x: undefined}
```

call forget to pass argument to constructor method!
=> using undefined

Class field / assignements from constructor

```
> class A {
      x; y; // cf also in ctor
      constructor(x) { this.x = x; }
  }
```

declare class with 2 fields: x, y

initialise only x

```
< undefined
```

```
> a = new A(1)
```

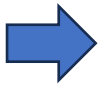
```
< ▶ A {x: 1, y: undefined}
```

field y exist, but not assign from constructor

Comparison with TypeScript ... short syntax for field declaration + assignment

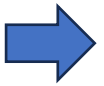
ONLY in TypeScript (yet)

```
class A {
   field4: string = '';

   constructor(public field1: string,
               private field2: string,
               private readonly field3: string,
               param: string) {
   }
}
```



// type script implicit equivalent

```
class A {
   public field1: string;
   private field2: string;
   private readonly field3: string;
   field4: string = '';

   constructor(paramField1: string,
               paramField2: string,
               paramField3: string,
               param: string) {
     this.field1 = paramField1;
     this.field2 = paramField2;
     this.field3 = paramField3;
   }
}
```



// JavaScript equivalent

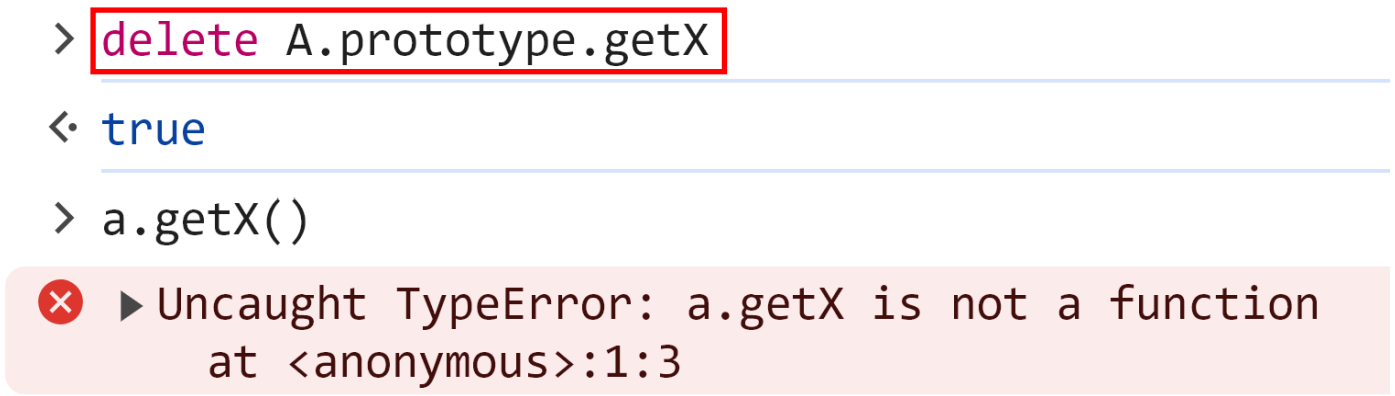
```
class A {
   field1;
   field2;
   field3;
   field4 = '';

   constructor(paramField1,
               paramField2,
               paramField3,
               param) {
     this.field1 = paramField1;
     this.field2 = paramField2;
     this.field3 = paramField3;
   }
}
```

Class Method

```
> class A {
      x; y;
      getX() { return this.x; }
  }
```

```
> a = new A()
◂ ▶ A {x: undefined, y: undefined}
> a.x = 1
◂ 1
> a.getX()
◂ 1
```

Class are just like prototype
=> Class are not immutable
proof: removing declared method via prototype



Class <-> prototype ...
methods can be added at runtime

```
> A.prototype.f = function() { console.log("call A.f"); return this.x; }
< ƒ () { console.log("call A.f"); return this.x; }

> a.x = 1
< 1
> a.f()
  call A.f
< 1
```

calling newly created method on previous existing object instance
=> works !

"undefined is not a function"

```
> delete A.prototype.f
<· true
> a.f()
✕ ▶Uncaught TypeError: a.f is not a function
      at <anonymous>:1:3
```

```
> af = a.f
<· undefined
> af()
✕ ▶Uncaught TypeError: af is not a function
      at <anonymous>:1:1
> undefined()
✕ ▶Uncaught TypeError: undefined is not a function
      at <anonymous>:1:1
```

Corresponding Errors in Java

NullPointerException
ArrayIndexOutOfBoundException

Corresponding Errors in C,C++,..

SegmentationFault
Bus Error
Memory Access Error

Method as field per instance !??
(instead of per prototype)

```
> class B {
      z;
      getZ = function() { return this.z; } // field, not per prototype!
  }
```

```
> b1 = new B();  b2 = new B();
```

instanciating several objects
=> call several times "this.getZ = function() {..} "

```
> b1.getZ === b2.getZ
< false
```

redundant (duplicated) functions instances!
may have huge performance/memory impact

Class property : get | set

```
> class A {
      x = 1;
      get xProp() { console.log('call get A.xProp', this.x); return this.x; }
      set xProp(value) { console.log('call set A.xProp', value); this.x = value; }
  }
```

```
> a = new A()
<· ▶ A {x: 1}
> a.x
<· 1
```

```
> a.xProp
call get A.xProp 1
<· 1
> a.xProp = 2
call set A.xProp 2
<· 2
> a.x
<· 2
```

call property getter

call property setter

Class inheritance

```
> class B extends A { z; }
< undefined
> B
< class B extends A { z; }
> b = new B();
< ▶ B {x: undefined, y: undefined, z: undefined}
```

using sub-class and methods override is NOT (yet?) idiomatic code in JavaScript

but works "fine"
(without type check / immutability constraint)

Class static : method | field | get | set

```
> class A {
      static x = 1;
      static foo() { console.log('A.foo', this.x); }
      static get xProp() { console.log('get A.xProp', this.x); return this.x; }
      static set xProp(value) { console.log('set A.xProp', value); this.x = value; }
  }
```

```
< undefined
```

```
> A.foo()
  A.foo 1
< undefined
```

call static function

```
> A.x
< 1
```

get/set static field

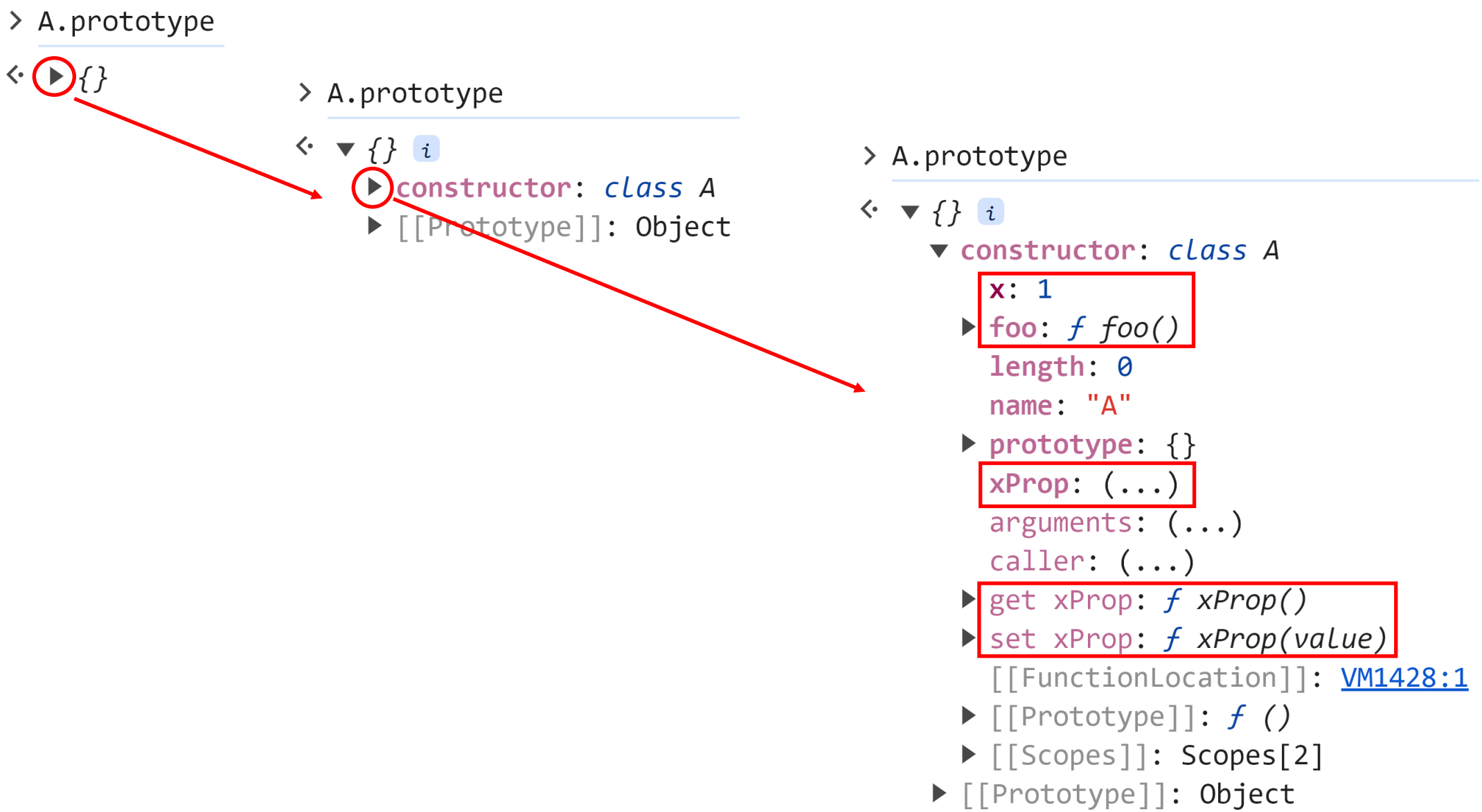
```
> A.xProp
  get A.xProp 1
< 1
```

call static property getter

```
> A.xProp = 2
  set A.xProp 2
< 2
> A.x
< 2
```

call static property setter

Class statics : stored on prototype



OK there are "**class**",

but class are not "**Type**"
and still **NO type-checking** at compile-time

Mostly on any error, you will get :

Exception "undefined is not a function"

Type checking ?
at "Compile Time" ??

add optional " : <typeDeclaration>" to variables

TypeScript = superset of JavaScript langage

a JavaScript file is a TypeScript file
(valid grammar, but maybe invalid type check)

Edit Js to add ": <type>" to variables
Rename file ".js" to ".ts" TypeScript

": <type>" is not valid JavaScript

```
> let a: string = 'test'
Uncaught
SyntaxError: Unexpected token ':'
```

valid in TypeScript

a.ts

```ts

let a: string = 'text';
console.log('Hello TypeScript', a);

```

invalid.js

```
1
2  let a : string = 'text';
```

Types are not supported by current JavaScript version

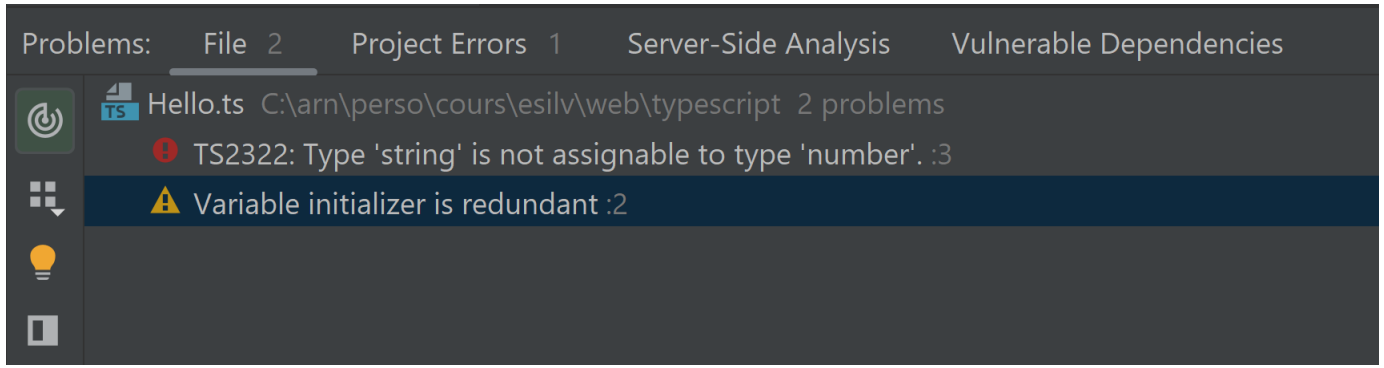
Change JavaScript language version to Flow Alt+Maj+Entrée More actions... Ctrl+1 Alt+Entrée

Compile Time checking ...

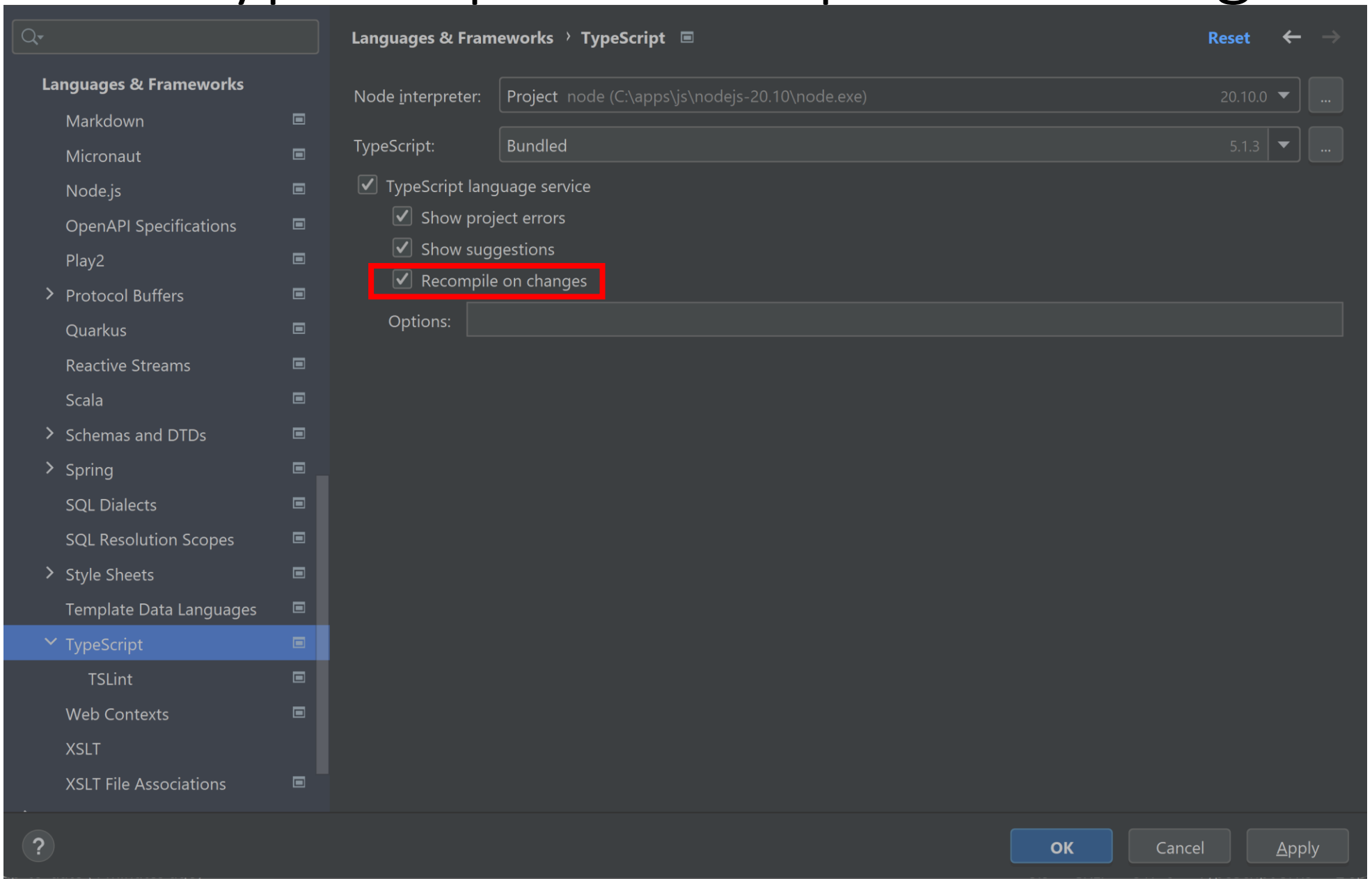
Hello.ts

```
1
2   let err : number  = 12; // number
3   err = 'changed to text'; // compile error in TypeScript
```

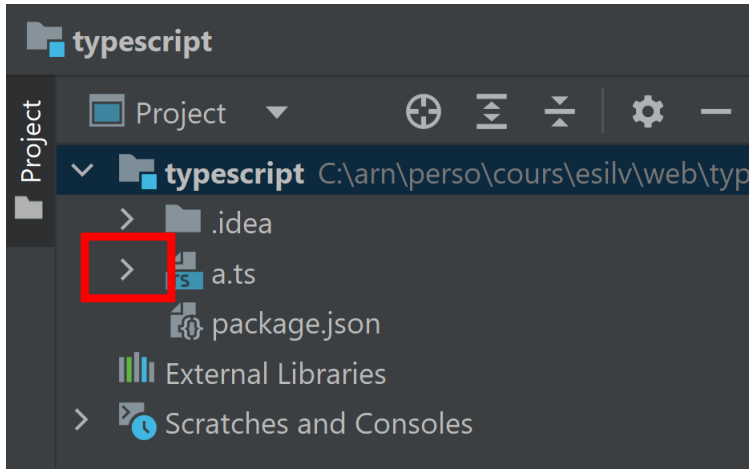
valid in untyped JavaScript
but not in TypeScript



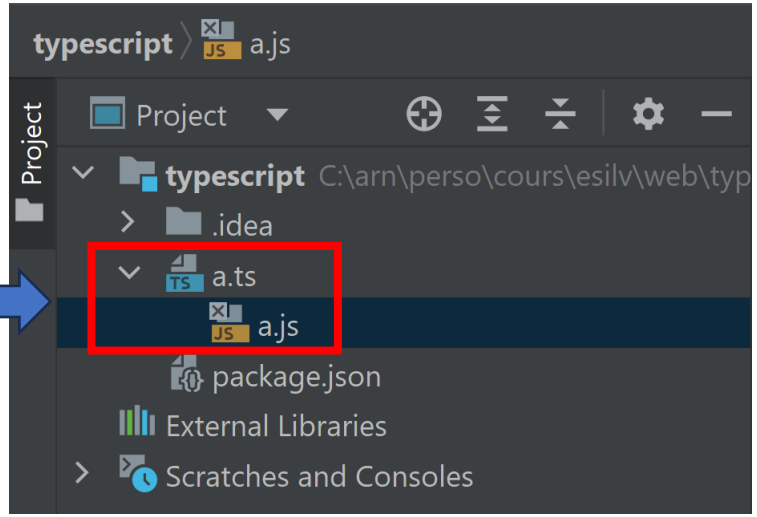
IntelliJ TypeScript : Recompile on changes



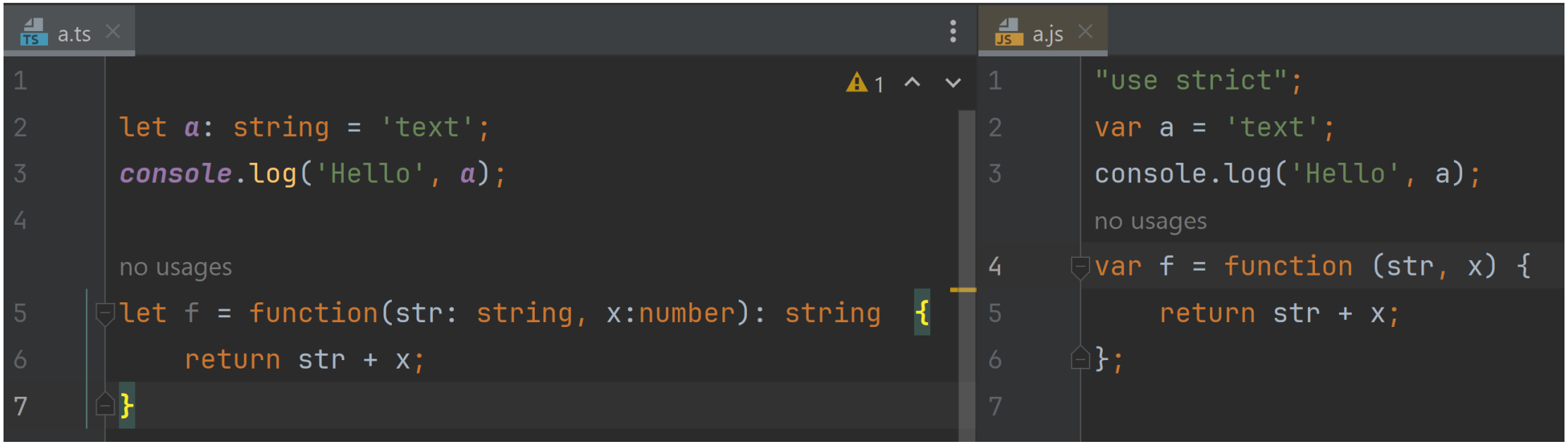
Js file generated from Ts file



expand to see generated file



Transpile TypeScript
= generate JS by mostly removing
": <typeDeclaration>"



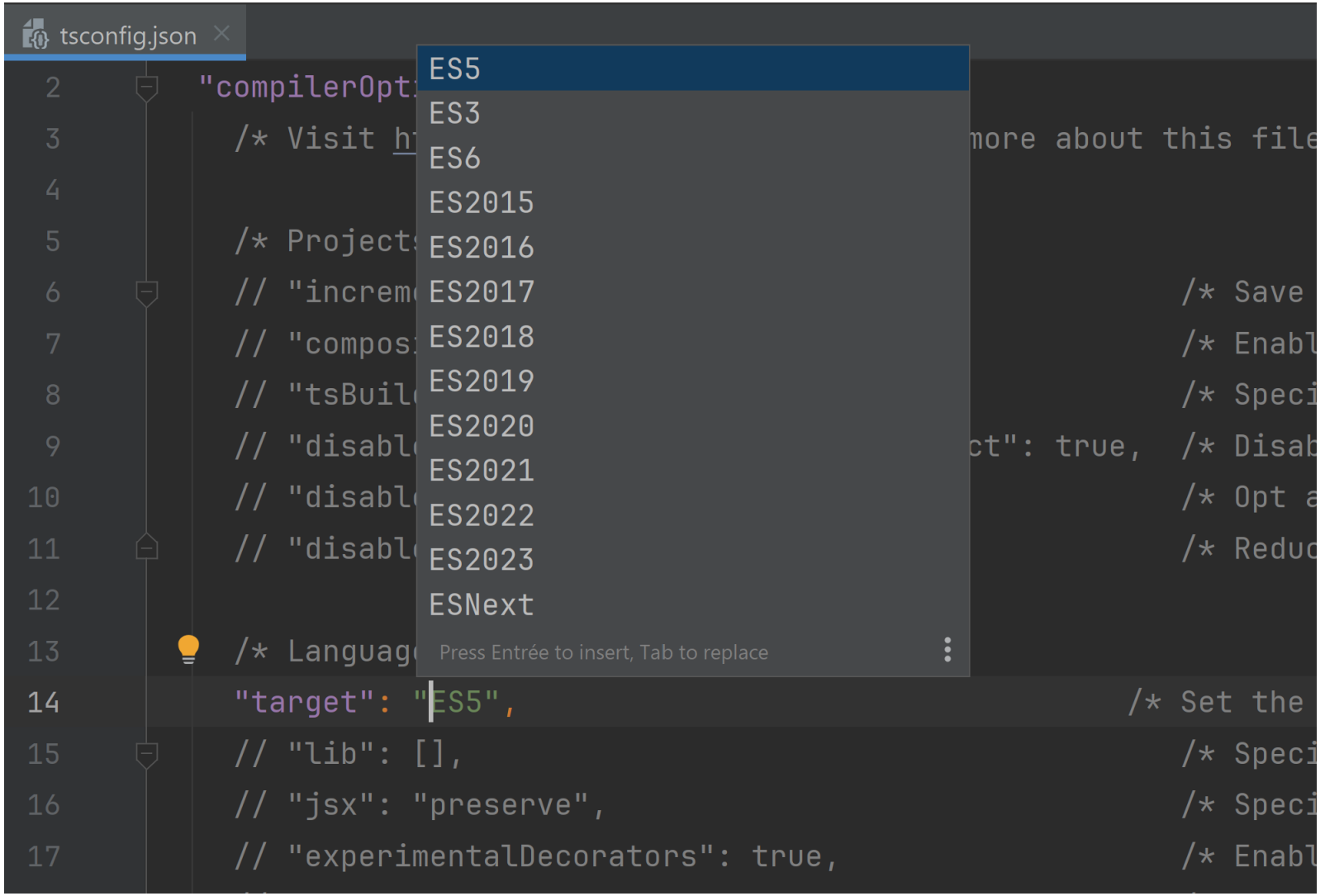
Tsc (=TypeScript Compiler)

```
$ tsc --help
tsc: The TypeScript Compiler - Version 5.3.3

COMMON COMMANDS

  tsc
  Compiles the current project (tsconfig.json in the working directory.)

  tsc app.ts util.ts
  Ignoring tsconfig.json, compiles the specified files with default compiler options.

  tsc -b
  Build a composite project in the working directory.

  tsc --init
  Creates a tsconfig.json with the recommended settings in the working directory.
```

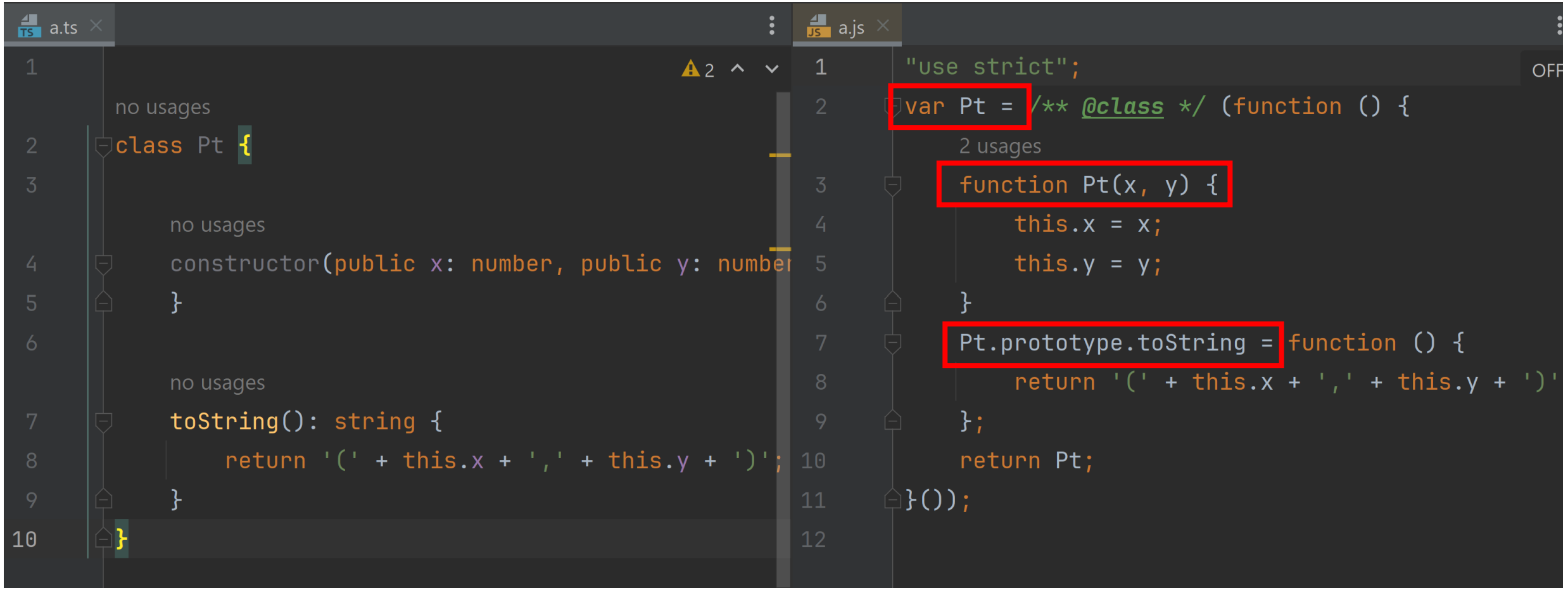
tsc --init

```
$ tsc --init

Created a new tsconfig.json with:

  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true


You can learn more at https://aka.ms/tsconfig
```

tsconfig.json using ES5 … ES2023



Example Emulating class with ES5



JavaScript evolve (from TypeScript ESNext)
… example: have class since ES2015, after TS had

Pt.ts

```ts
no usages
class Pt {

    no usages
    constructor(public x: number, public y: number) {
    }

    no usages
    toString(): string {
        return '(' + this.x + ',' + this.y + ')';
    }
}
```

Pt.js

```js
"use strict";
no usages
class Pt {
    no usages
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    no usages
    toString() : string  {
        return '(' + this.x + ',' + this.y + ')';
    }
}
```

JavaScript has weird "var" visibility

JavaScript has weird "var" visibility

"use strict"; // at first line

But still weird

JavaScript has weird "var" visibility

"use strict"; // at first line

use Immediatly Invoked Anonymous Function

But still weird

JavaScript has weird "var" visibility

"use strict"; // at first line

use Immediatly Invoked Anonymous Function

use "let" or "const" instead of "var"

use TypeScript

in "Standard Langages"
1/ declare, then 2/ init, then- 3/ use
with scope = enclosing { block }

block-decl.js

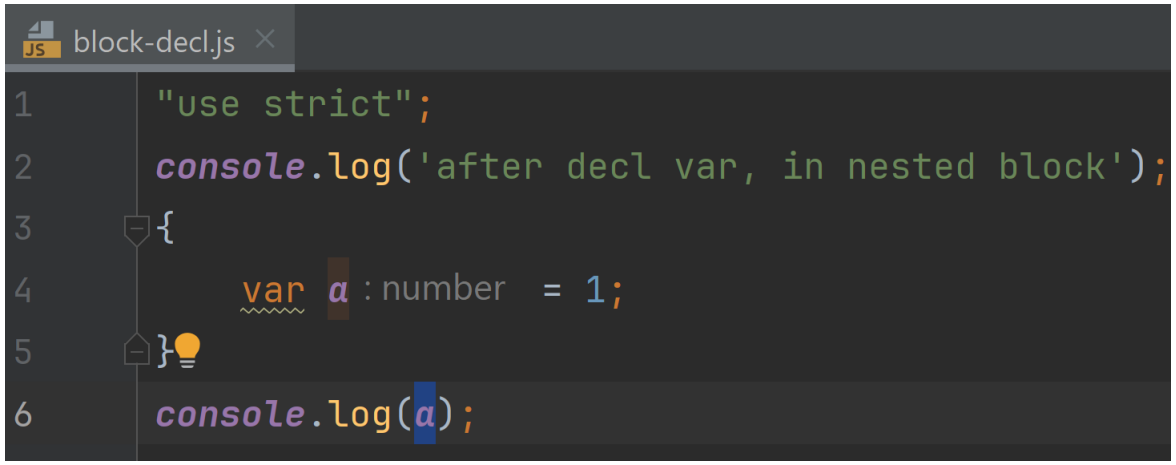
```
1  "use strict";
2  console.log('after decl let, in same block');
3  {
4      let a : number  = 1;
5      console.log(a); // OK, standard case
6  }
```

Debug: block-decl.js

Debugger | Debugger Console | Process Console

```
after decl let, in same block
1
```

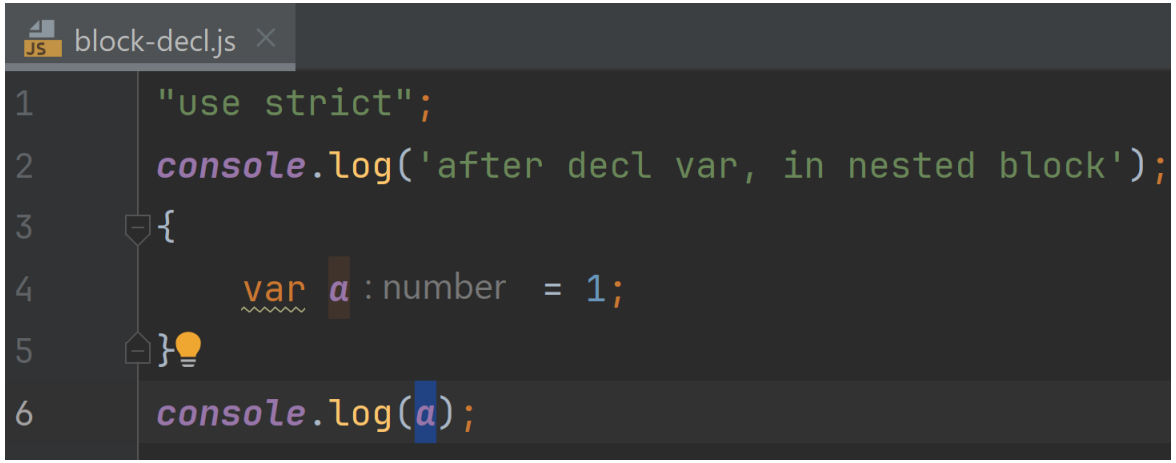
JavaScript has weird "var" visibility

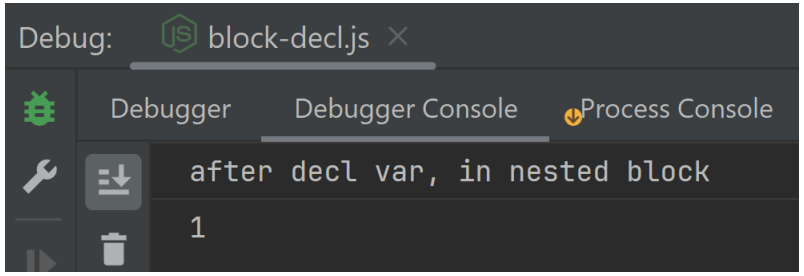


What do you think ??

a/ does not "evaluate" / "run" / "compile"
b/ throws exception
c/ print "undefined"
d/ print "1"

Answer: d/ print 1
the scope of visibility if the enclosing function
(not the enclosing block)

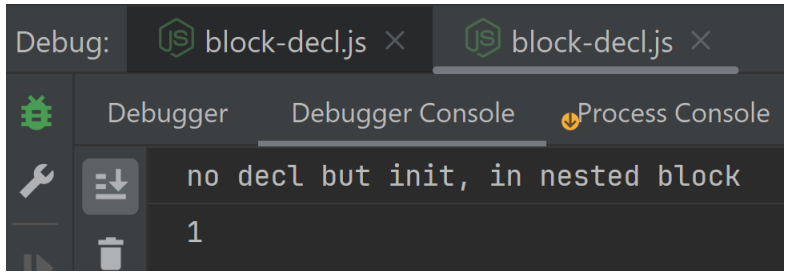




without "use strict" ... NO need to decl "var"

block-decl.js

```
1  // NO "use strict";
2  console.log('no decl but init, in nested block');
3  {
4      a = 1;
5  }
6  console.log(a);
```



with "use strict";

block-decl.js

```
1  "use strict";
2  console.log('no decl but init, in nested block');
3  try {
4      a = 1;
5  } catch(e) {
6      console.log('err', e)
7  }
```

Debug: block-decl.js × block-decl.js ×

Debugger | Debugger Console | Process Console

```
no decl but init, in nested block

err
> ReferenceError: a is not defined at Object.<anonymous> (C:\arn\perso\cours\e
  (node:internal/modules/cjs/loader:1376:14) at Module._extensions..js (node:i
  (node:internal/modules/cjs/loader:1207:32) at Module._load (node:internal/mo
  (node:internal/modules/run_main:135:12) at node:internal/main/run_main_modul
  node:internal/main/run_main_module:28:49", message: "a is not defined"}
```

even stranger:
use before declare in nested block

block-decl.js

```
1  "use strict";
2  console.log('before decl var, in nested block');
3  console.log(a);
4  {
5      var a : number  = 1;
6  }
```

Debug: block-decl.js

Debugger | Debugger Console | Process Console

```
before decl var, in nested block
undefined
```

other strange "var" visibility

block-decl.js

```
1  "use strict";
2  console.log('after decl var, in nested block');
3  {
4      var a : number  = 1;
5  }
6  console.log(a);
```

Debug: block-decl.js

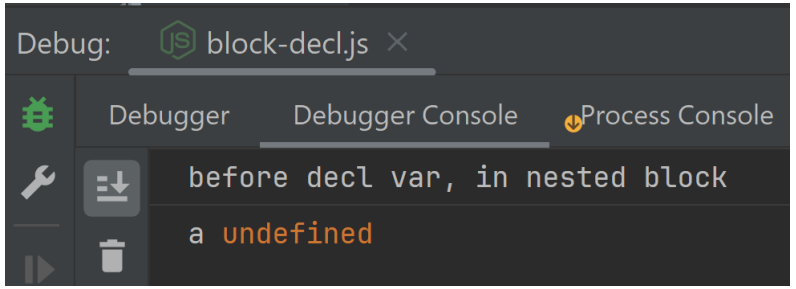
Debugger | Debugger Console | Process Console

```
after decl var, in nested block
1
```

JavaScript has weird "var" visibility

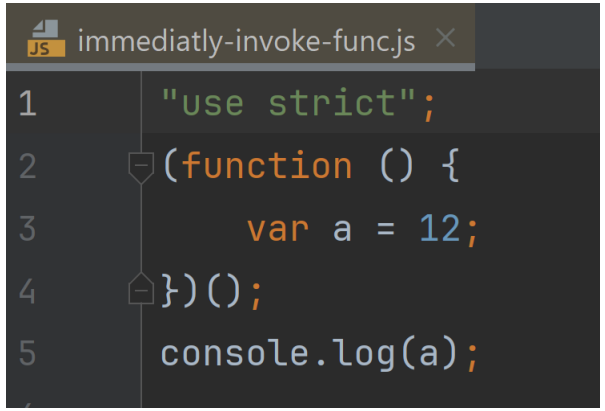
block-decl.js

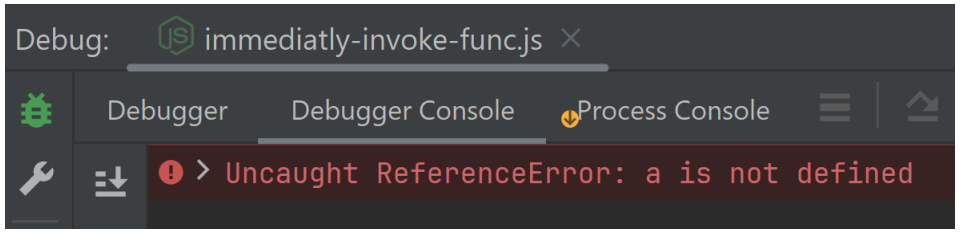
```
1  "use strict";
2  console.log('before decl var, in nested block');
3  console.log(a);
4  {
5      var a : number  = 1;
6  }
```



declared "ok"
BUT not initialized yet => undefined !

work around for scope :
"Immediatly Invoked Anonymous Function"





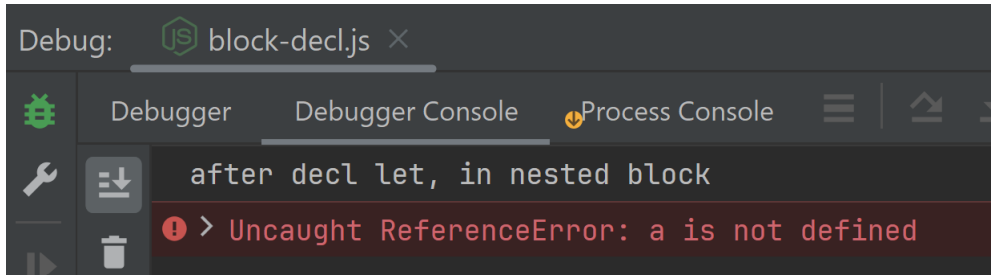
"let" | "const" ... as "var" replacement

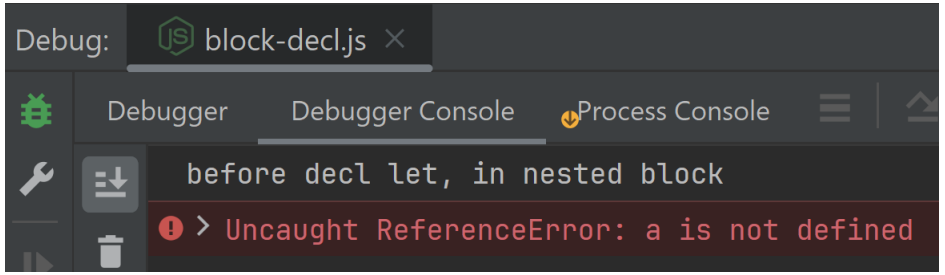
block-decl.js

```
1  "use strict";
2  console.log('after decl let, in nested block');
3  {
4      let a : number  = 1;
5  }
6  console.log(a);
```

block-decl.js

```
1  "use strict";
2  console.log('before decl let, in nested block');
3  console.log(a);
4  {
5      let a : number  = 1;
6  }
```





"let" (or "const") in same block

block-decl.js

```
1   "use strict";
2   console.log('before decl let, in same block');
3   {
4       console.log(a);
5       let a : number  = 1;
6   }
```

Debug: block-decl.js

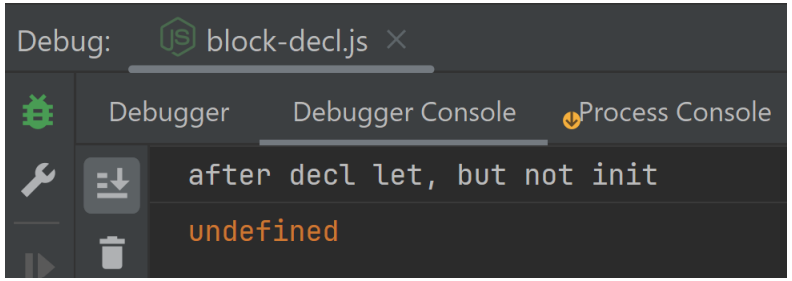
Debugger | Debugger Console | Process Console

```
before decl let, in same block
> Uncaught ReferenceError: Cannot access 'a' before initialization
```

not init (ok in JS !)
but does not compile in TypeScript

block-decl.js

```
1  "use strict";
2  console.log('after decl let, but not init');
3  {
4      let a;
5      console.log(a);
6      a = 1;
7  }
```



everybody hate JavaScript, for million reasons,
but it does evolve

100 reasons you need TypeScript
and will probably love it

Spoiler Alert

TypeScript is NOT "more efficient" than JS

... it is just typed-checked at Compile-Time
then translated to JS

at Runtime, you still have poor old
Node.js or Web Browser

TypeScript Types

Primitive Types (for values) : any, null, undefined, boolean, number, string

Composite Types:

" **|** " Union (choice between types)
" **&** " combination of type constraint
"**readonly**" type modifier
Array
Interfaces : type constraint on existence of field / partial field in Object
" **?** " Optional Field Type (for accepting "undefined")
Function, Class, Templates, etc.

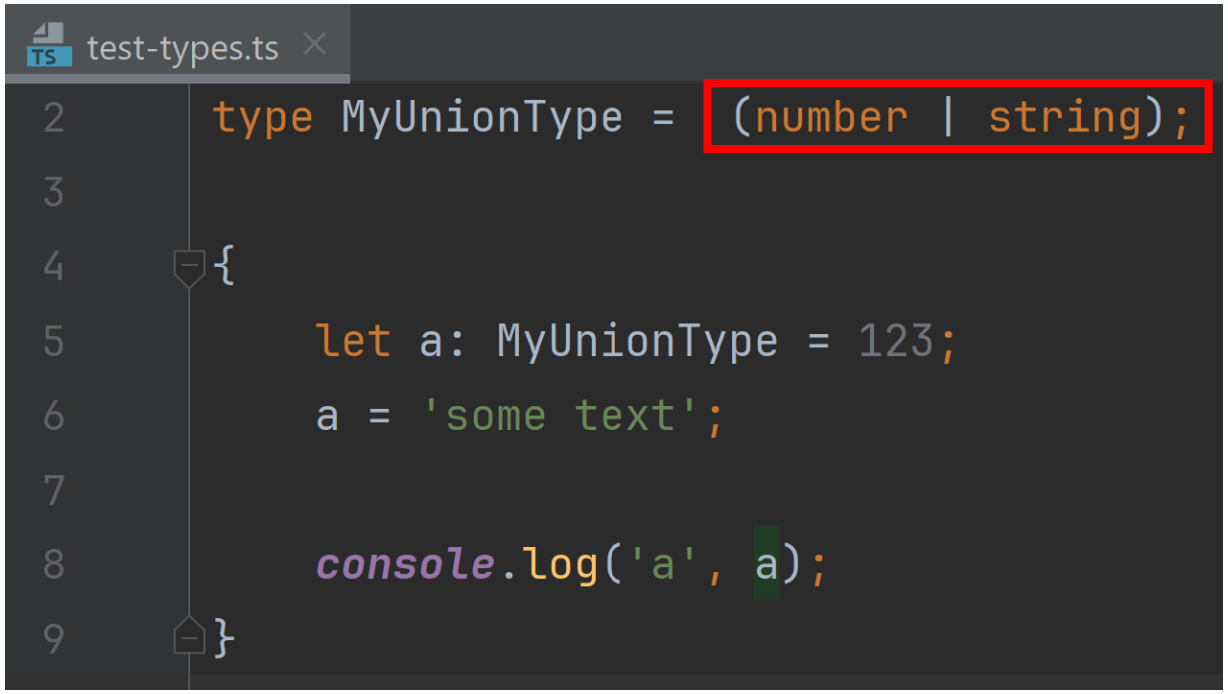
any ?

any ... as name implies
= No Type checking !

To use TEMPORARILY while porting JS to Ts

Good Ts program(er)s should never use "any"

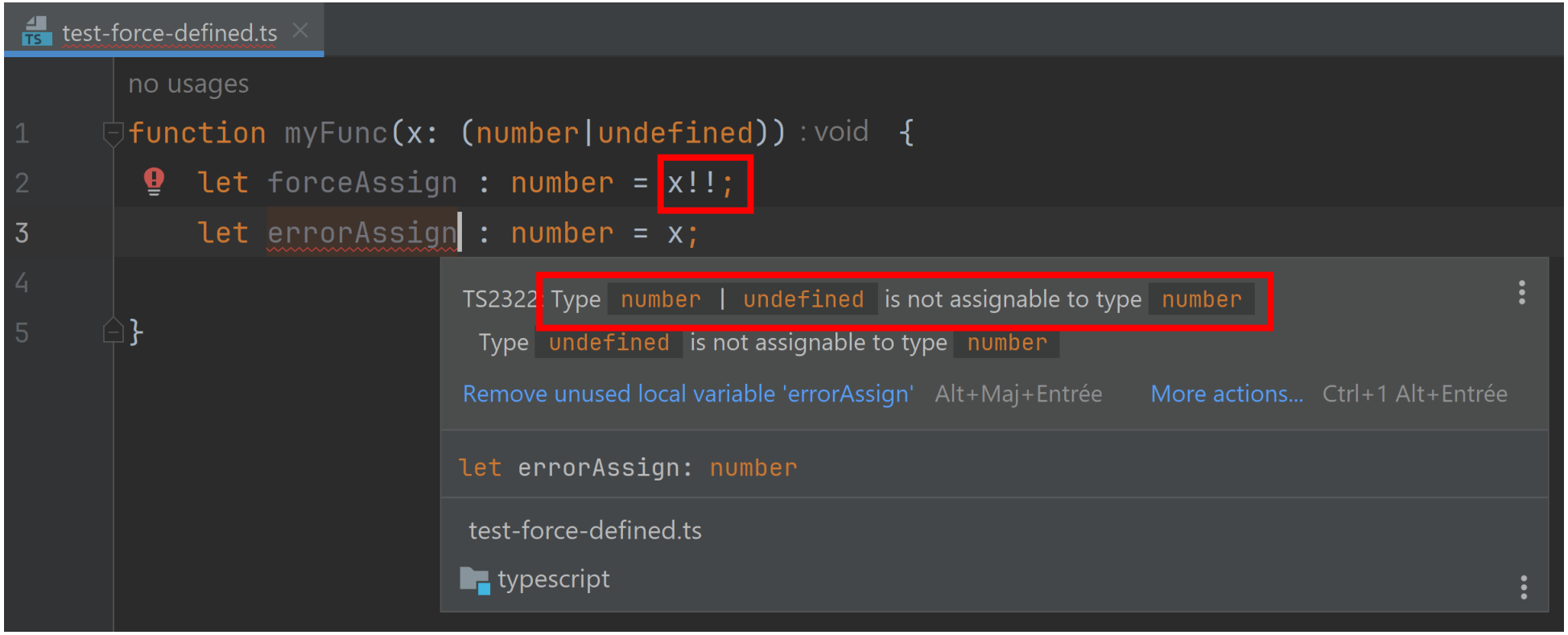
Union Type


test-types.ts
type MyUnionType = (number | string);
{
let a: MyUnionType = 123;
a = 'some text';
console.log('a', a);
}

Example usages
": Type |undefined" for un-initialized field

```
class A {

    field1 : string|undefined;   // OK but ugly ... why undefined type? because implicit undefined value !

    field2 : string;     // compile ERROR !!    "undefined is not assignable to string"

    field3: string = 'text';  // OK field assigned to string value

    field4 = 'text';  // OK field assigned, so field type to implicitly infered to ":string"

    field5: string;  // ok with assignment next in constructor

    constructor() {
      this. field5 = 'text';
    }
```

Force type-check with "!!"



Duck Typing

duck-typing.ts

```
2 usages
interface Duck { name : string };

{
    let obj : {name: string}  = { name: 'abc' };

    // obj is a Duck !
    let duckObj : Duck = obj;  // explicit typed, from implicit assign
    let duckObj2 : Duck  = <Duck> obj; // implicit typed, explicit coerce
}
```

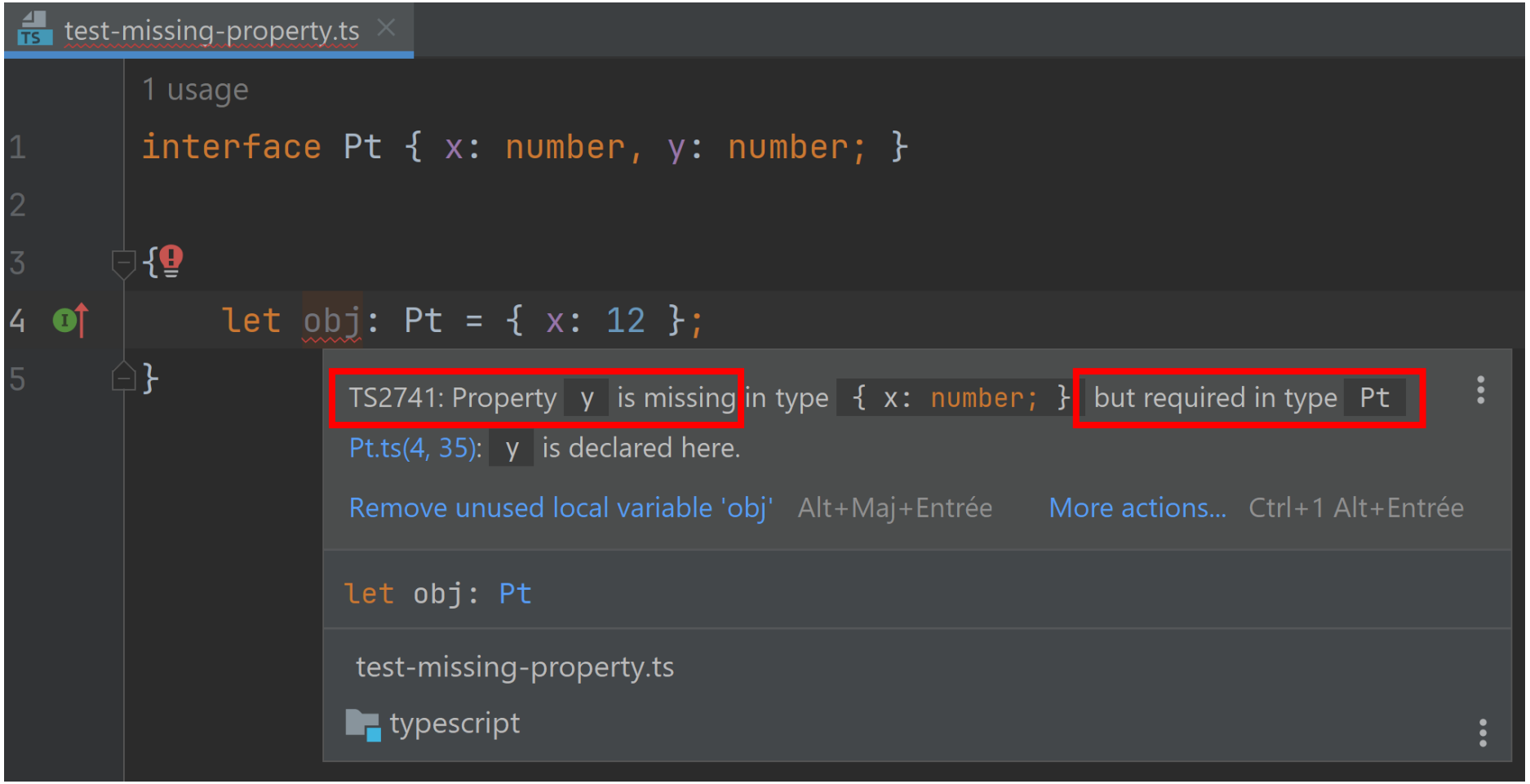
Structural Interface Typing, Combine Partial

test-interface.ts

```
1   interface Ix { x: number; }
    1 usage
2   interface Iy { y: number; }
3   type Ixy = Ix & Iy;
4   type Pt = {  x: number; y: number }; // equivalent to Ix & Iy
5
6   {
7       let objX                = { x: 1 };  // type inferred = { x:number }
8       let objX2 : Ix = objX;  // implicit type "cast" ok
9       let objY        = { y: 1};
10      let obj: Pt = { ...objX, ...objY};
11      console.log(obj);
12  }
```

= type inferred (displayed by IntelliJ), NOT edited in file

Type Checking "missing property" in Interface



TypeScript...

TypeScript is AWESOME ！

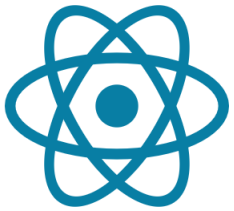
allow to write very concise functional / json structured code

What about JSX / TSX ?

X for (xml) eXtension ?

React
… JS functions for rendering html

https://react.dev/

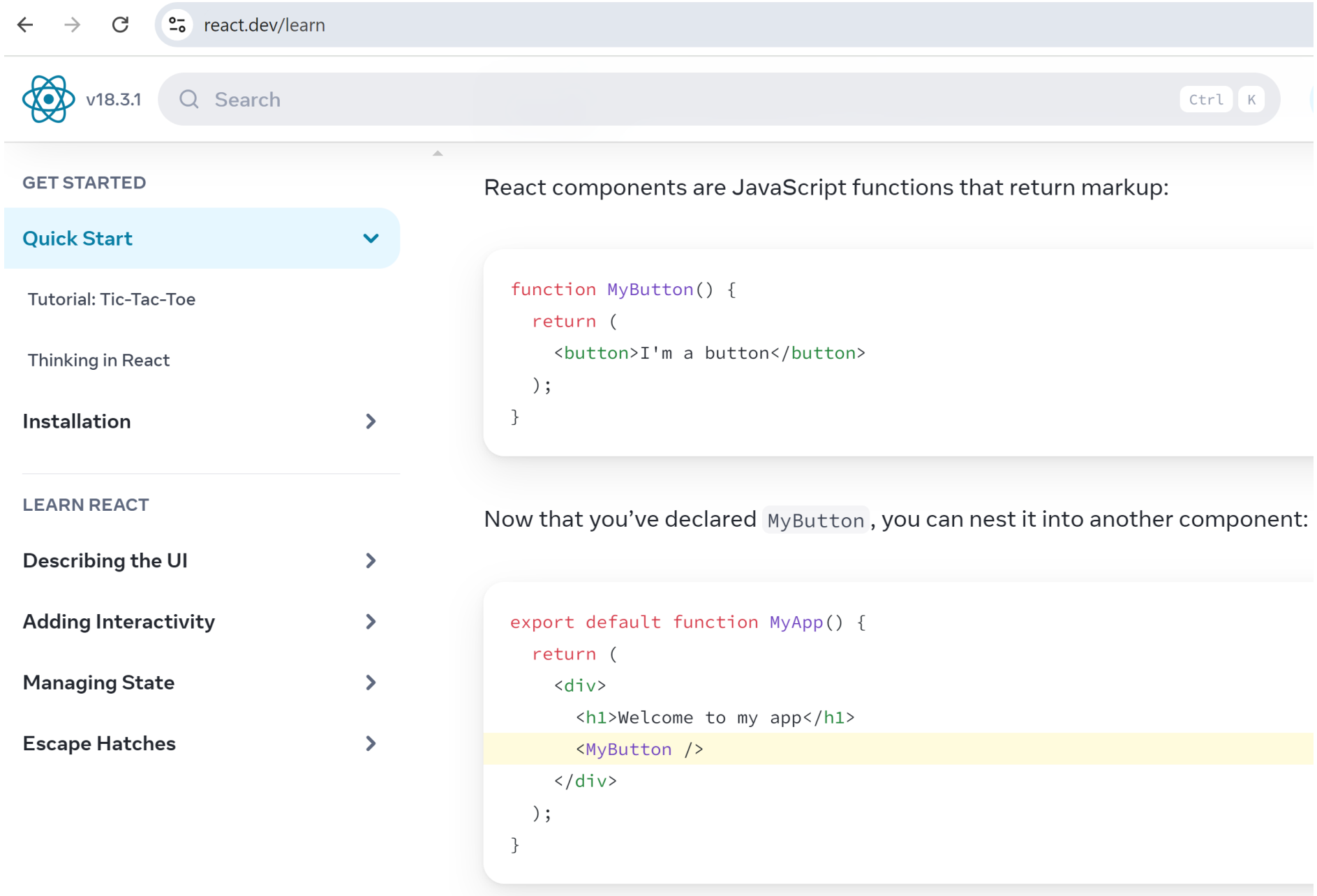


React

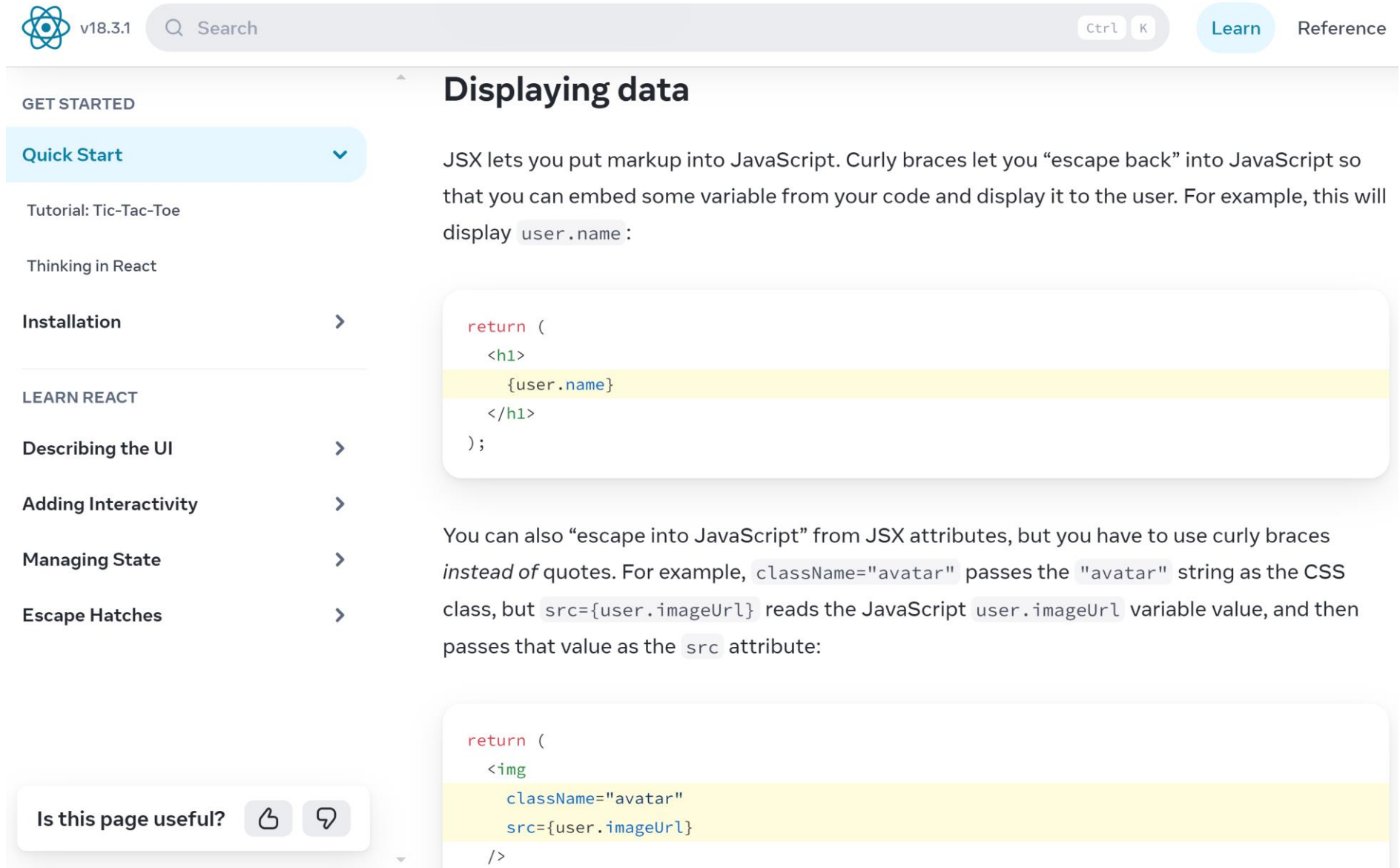
The library for web and native user interfaces

Learn React | API Reference

function() { return (<html> </html>); }

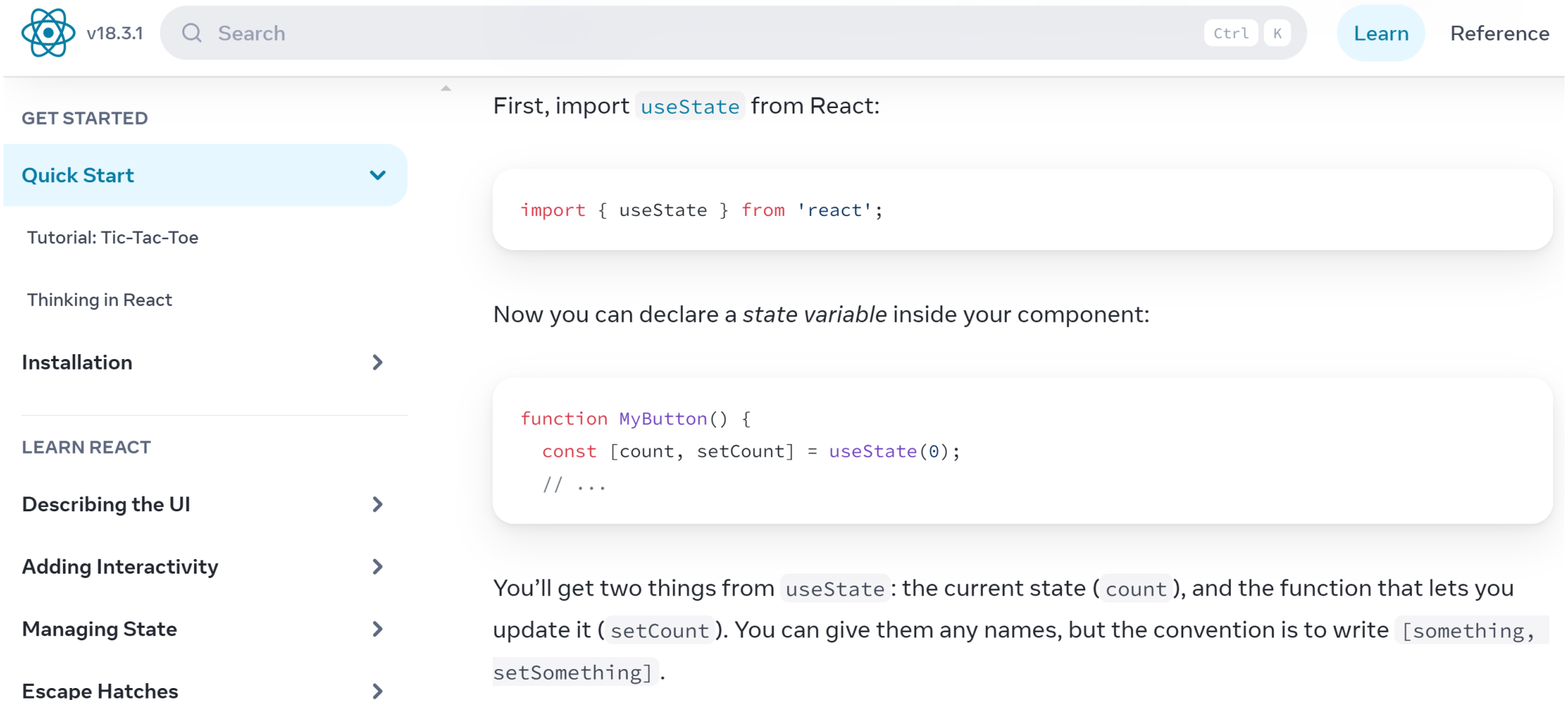


<html> with templated JS { someJS() } </html>



Not valid JavaScript / TypeScript !
need to be "translated" to JavaScript

Function instead of "Class" object instances ! where to store field state?



And How to pass input values from parent to child?


```
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}
```

The information you pass down like this is called *props*. Now the `MyApp` component contains the `count` state and the `handleClick` event handler, and *passes both of them down as props* to each of the buttons.

Finally, change `MyButton` to *read* the props you have passed from its parent component:

```
function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}
```

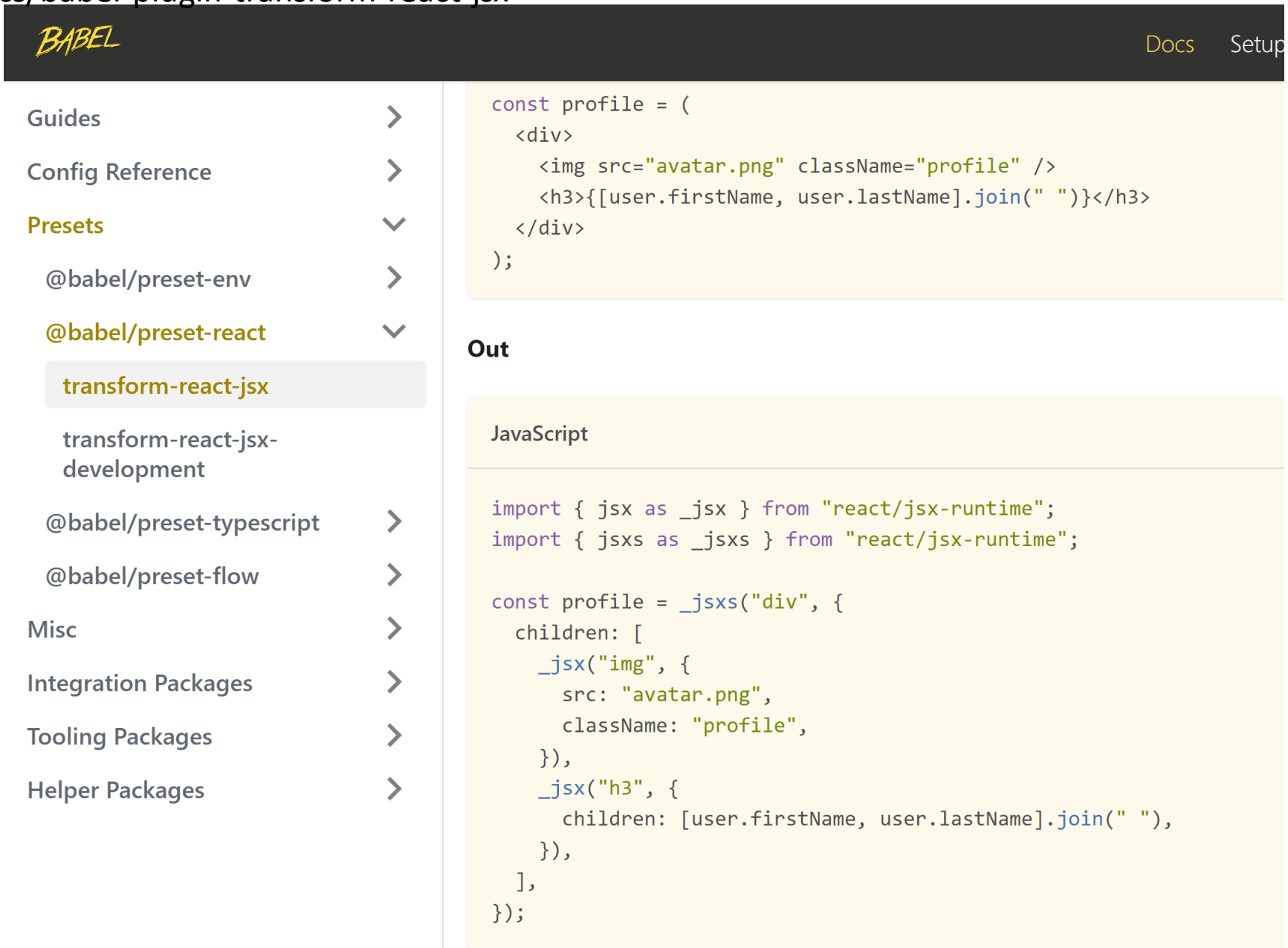
… the start of the ending !

inventing other workaround: React Redux
& many more … MobX, React State,..

Babel: translate JSX to JS
(with react/jsx-runtime functions calls)

Babel Translated Code

https://babeljs.io/docs/babel-plugin-transform-react-jsx



Why not having N x 2 files for using 2 different langages: Html != TS

When developping with N small components
you may need to switch between 2 x N file : comp1.ts, comp1.html, comp2.{js|html}, ..

example in Angular: $ ng generate component foo
=> get 2 files: foo.component.ts + foo.component.html

You may also inline the html in the ts file (template='<h1>..' instead of templateUrl='./file-path.ts')

```
@Component(selector='foo'
   template=`
      <h1>... some long html</h1>
     multi-lines
   `
)
```

You may also declare several components (+ associated template) in a single .ts file

Web Editor Support

Fortunatly, good editors
 (IntelliJ Ultimate or IntelliJ WebStorm, VisualCode, etc..)

support contextual

 TypeScript edit features within { code() }
& Html edit features within <element> .. </element>

Integrating React in a Web Page [1/2]

Step 1: Set up a modular JavaScript environment

A modular JavaScript environment lets you write your React components in individual files, as opposed to writing all of your code in a single file. It also lets you use all the wonderful packages published by other developers on the npm registry—including React itself! How you do this depends on your existing setup:

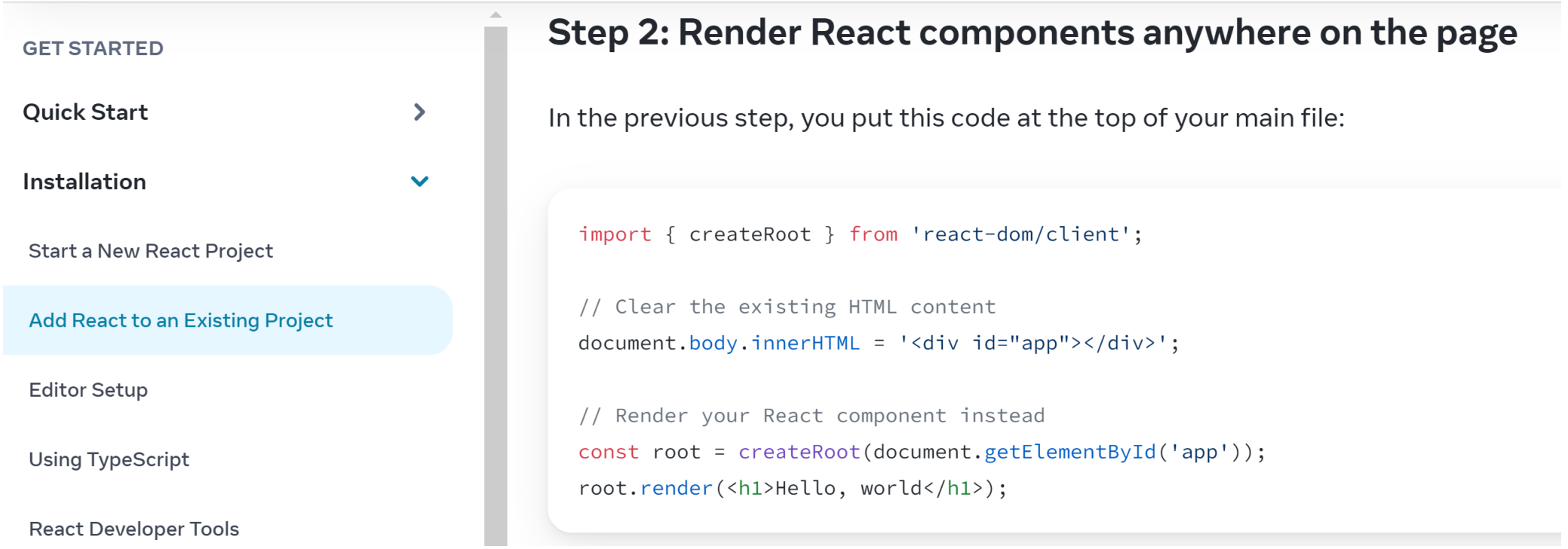
- **If your app is already split into files that use `import` statements,** try to use the setup you already have. Check whether writing `<div />` in your JS code causes a syntax error. If it causes a syntax error, you might need to transform your JavaScript code with Babel, and enable the Babel React preset to use JSX.
- **If your app doesn't have an existing setup for compiling JavaScript modules,** set it up with Vite. The Vite community maintains many integrations with backend frameworks, including Rails, Django, and Laravel. If your backend framework is not listed, follow this guide to manually integrate Vite builds with your backend.

To check whether your setup works, run this command in your project folder:

Terminal Copy

```
npm install react react-dom
```

Integrating React in a Web Page [2/2]



JS -> JSX, TypeScript -> TSX

Is this page useful?

Terminal Copy

```
npm install @types/react @types/react-dom
```

The following compiler options need to be set in your `tsconfig.json`:

1. `dom` must be included in `lib` (Note: If no `lib` option is specified, `dom` is included by default).
2. `jsx` must be set to one of the valid options. `preserve` should suffice for most applications. If you're publishing a library, consult the `jsx` documentation on what value to choose.

TypeScript with React Components

Note

Every file containing JSX must use the `.tsx` file extension. This is a TypeScript-specific extension that tells TypeScript that this file contains JSX.

Writing TypeScript with React is very similar to writing JavaScript with React. The key difference when working with a component is that you can provide types for your component's props

Take Away

JavaScript is a scripting langage (not suited for large code base app)
but is the de-facto standard Runtime for Web

TypeScript is the standard developper langage for generating JavaScript, with type checks

JSON is the standard data format for Http Rest API

JSX/TSX are eXtensions of JS/TS for React
... not a standard, only a fashion as of 2024.
other frameworks exist: Angular or VueJs are richer/simpler alternatives for data-binding + routes + ...

Next Steps

- JavaScript, TypeScript, JSON, TSX
- Interactions between Html (DOM) / Https / JavaScript
- Angular Web Framework
- Building Rest JSON API, and SPA Web application
- ..