

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

1995

1997

2005

2009

2012

2015



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..)	TypeScript birth of Node.JS server-side JS development ecosystem (npm, grunt, gulp, ..)	ECMAScript 6 (=2015) class, arrow, modules, ...
---	---	--	---	---	---

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

<https://jsisweird.com/>



even for me

A screenshot of a web browser window with the URL `jsisweird.com` in the address bar. The page has a yellow header and a dark blue footer. The main content area displays a large white text message: **You got 12 out of 25 correct!**. Below this, a dark blue box contains a question and its answer. The question is **1. true + false**. The output is `1`, and the user answered `1`. A green button says **You got it right!**. Below this, a note states: **According to the [ECMAScript Language Specification](#), the two boolean values are type coerced into their numeric counterparts.**. At the bottom, there is some code: `Number(true); // -> 1`, `Number(false); // -> 0`, and `1 + 0; // -> 1`.

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

JavaScript = un-maintainable code !!

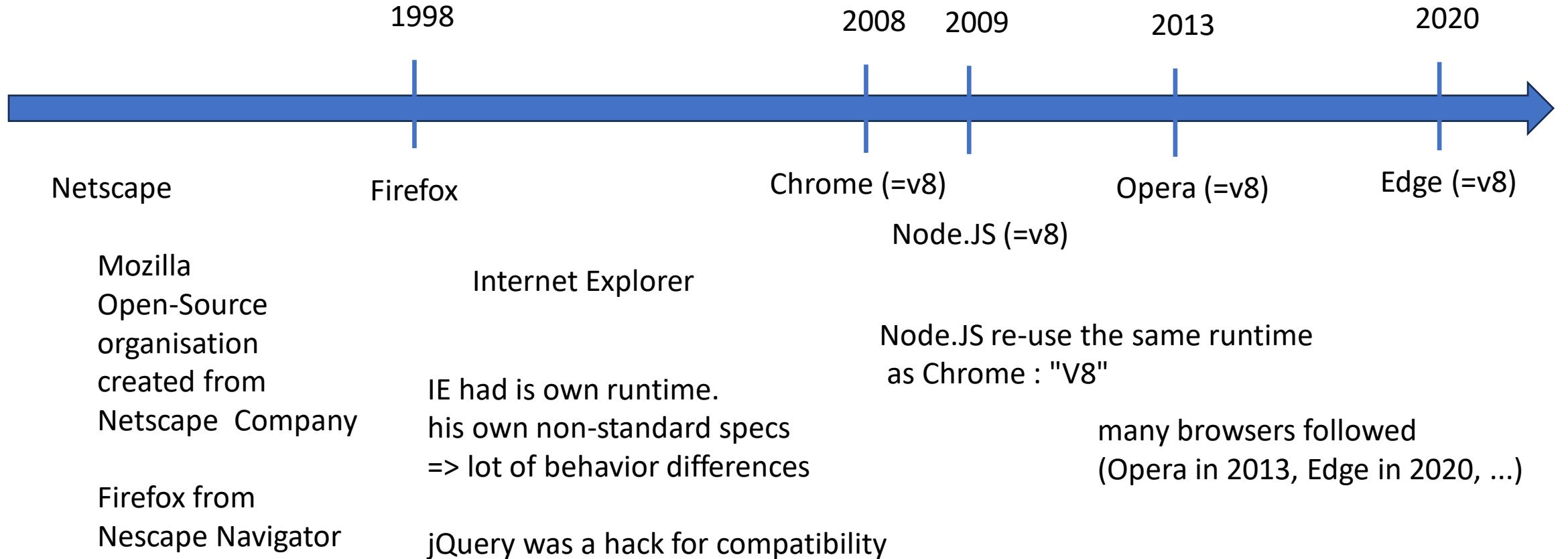
NO Type !!

- .. 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



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
1  let obj :{} = { a:null, b:undefined, c:12.3, d:true,
2    e: 'text with ', \ and \n',
3    f:"text with \", ' and \n",
4    g: [1, 2],
5    h: { a:1, b:2}
6  }
7  let objJson :string = JSON.stringify(obj);
8  console.log('json stringify:', objJson);
9
10 let objJsonIndented :string = JSON.stringify(obj, replacer: null, space: ' ');
11 console.log('json stringify indented:', objJsonIndented);
12
13 let objCopy = JSON.parse(objJson);
14 console.log('objCopy', objCopy)
```

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

The screenshot shows a debugger interface with the following details:

- Title Bar:** Debug: test-json.js
- Toolbar:** Debugger (selected), Debugger Console, Process Console, and several other icons.
- Left Sidebar:** A vertical sidebar with various icons corresponding to the toolbar items.
- Output Area:** Displays the result of `JSON.stringify` and `.parse` operations.
 - `json stringify: {"a":null,"c":12.3,"d":true,"e":"text with \", ' and \n","f":"text with \", ', and \n","g":[1,2],"h":{"a":1,"b":2}}`
 - `json stringify indented:` followed by a multi-line JSON object with indentation.
 - `objCopy > Object {a: null, c: 12.3, d: true, e: "text with ", ' and \n", f: "text with ", ', and \n", ...}`

JavaScript object deep clone

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

Objects (\sim = 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 }
```

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

```
< ► {x: 1, y: 2}
```

```
> let x = 1, y = 2;
```

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

```
< undefined
```

```
> a = { x, y }
```

```
< ► {x: 1, y: 2}
```

```
> b = { x:1, y }
```

```
< ► {x: 1, y: 2}
```

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 ])
```



The screenshot shows the VS Code interface during a debug session. The title bar says "Debug: test-object-structuringjs". The toolbar has icons for Debugger, Debugger Console (which is selected), Process Console, and various navigation and search tools. The main area displays the output of three `console.log` statements:

- "message with structured" followed by an object with properties a, b, c, d, e, f, g.
- "message with a,b" followed by an object with properties a and b.
- "message with [a,b]" followed by an array with elements a and 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

set field undefined

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

delete field

Object De-Structuring to Variables

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

```
> let {x} = obj          extract property "x" from obj, assign to local variable x  
< undefined
```

```
> x
```

```
< 1
```

```
> let {z,y} = obj
```

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

```
< undefined
```

```
> y
```

```
< 2
```

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()

object param is undefined !

✖ ► Uncaught TypeError: Cannot destructure property 'a' of 'undefined' as it is undefined.

at f (<anonymous>:1:13)
at <anonymous>:1:1

> 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)
```

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

```
> arr
```

```
< ▶ (6) [1, 2, 3, 4, 5, 6]
```

```
> arr.splice(3, 1)
```

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

```
> arr
```

```
< ▶ (5) [1, 2, 3, 5, 6]
```

Array operations

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



Equivalent Code in Java

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

```
> arr
```

```
< ▶ (6) [1, 2, 3, 1000, 4, 5]
```

```
> arr.indexOf(1000) arr.indexOf(1000);
```

```
< 3
```

```
> arr.findIndex( x => x === 1000) no simple equivalent ?
```

```
< 3
```

```
> 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)
```

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

```
< ▶ (2) [4, 5]
```

```
> b = arr.map(x => x*2)
```

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

```
< ▶ (5) [2, 4, 6, 8, 10]
```

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

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

```
< 15
```

Array Spreading Operator

```
> let a = [ 3, 4 ]
```

```
> [ 1, 2, ...a, 5 ]
```

```
< ► (5) [1, 2, 3, 4, 5]
```

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)          call function without parameter "b" => using default
```

```
< 4
```

```
> f(1,1)          call function with explicit parameter override
```

```
< 2
```

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
```

Idiomatic "Strange" JS Code

destructuring param & default param

```
> function f({a, b} = {a:1, b:2}) { console.log('call f', { a, b}); }
```

param default value
object destructuring

```
> f()
```

```
call f ▶ {a: 1, b: 2}
```

```
> f({a:5})
```

```
call f ▶ {a: 5, b: undefined}
```

Functions are Objects

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

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

```
> obj = [...f ]
```

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

```
< ► {}
```

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

can assign field to "function object"

```
< 1
```

```
> f
```

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

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

```
> f.x
```

can get field from "function object"

```
< 1
```

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

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

> f === g
< false
```

Arrow (Lambda) Function

```
> let myFunc = (a,b) => { console.log('myFunc()', {a,b}); }           arrow function with code block
< undefined
-----  
> myFunc(1, 2)
myFunc() ▶ {a: 1, b: 2}
< undefined
-----  
> let myFuncPlus = (a,b) => a+b                               arrow function with expression
< undefined
-----  
> myFuncPlus(1, 2)
< 3
```

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; }
```

```
< f () { return this.x; }
```

```
> MyConstructor.prototype.getY = function() { return this.y; }
```

```
< f () { 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 objects)

Class in JavaScript !

```
> class A { x; y } define a class
```

```
< undefined
```

```
> 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);          new call constructor
```

```
< ▶ A {x: 1}
```

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

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

Class field / assignments 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)
```

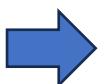
```
< ▶ A {x: 1, y: undefined}
```

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

```
> delete A.prototype.getX
```

```
< true
```

```
> a.getX()
```

✖ ► Uncaught TypeError: a.getX is not a function
at <anonymous>:1:3

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

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

ArrayIndexOutOfBoundsException

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() call static function

A.foo 1

< undefined

> A.x get/set static field

< 1

> A.xProp call static property getter

get A.xProp 1

< 1

> A.xProp = 2 call static property setter

set A.xProp 2

< 2

> A.x

< 2

Class statics : stored on prototype

```
> A.prototype
```

```
< ◀ { }
```

```
> A.prototype
```

```
< ▼ { } i
```

```
▶ constructor: class A  
▶ [[Prototype]]: Object
```

```
> A.prototype
```

```
< ▼ { } i
```

```
▶ constructor: class A
```

```
x: 1  
▶ foo: f foo()  
length: 0
```

```
name: "A"
```

```
▶ prototype: {}
```

```
xProp: (...)
```

```
arguments: (...)
```

```
caller: (...)
```

```
▶ get xProp: f xProp()
```

```
▶ set xProp: f xProp(value)
```

```
[[FunctionLocation]]: VM1428:1
```

```
▶ [[Prototype]]: f ()
```

```
▶ [[Scopes]]: Scopes[2]
```

```
▶ [[Prototype]]: Object
```



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

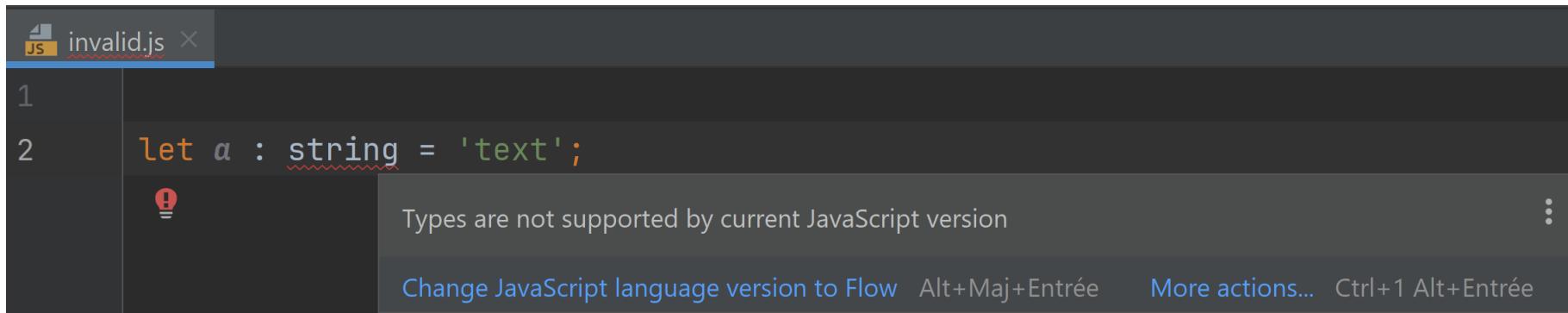
": <type>" is not valid JavaScript

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

valid in TypeScript



```
a.ts  
1  
2 let a: string = 'text';  
3 console.log('Hello TypeScript', a);
```

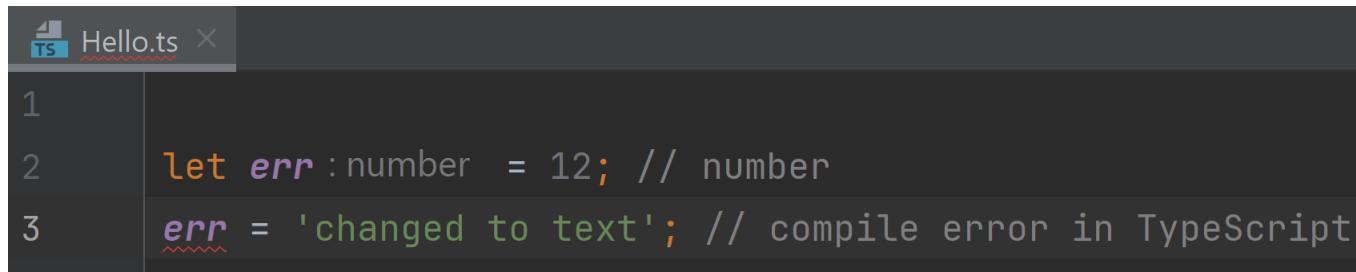


```
invalid.js  
1  
2 let a : string = 'text';
```

! Types are not supported by current JavaScript version

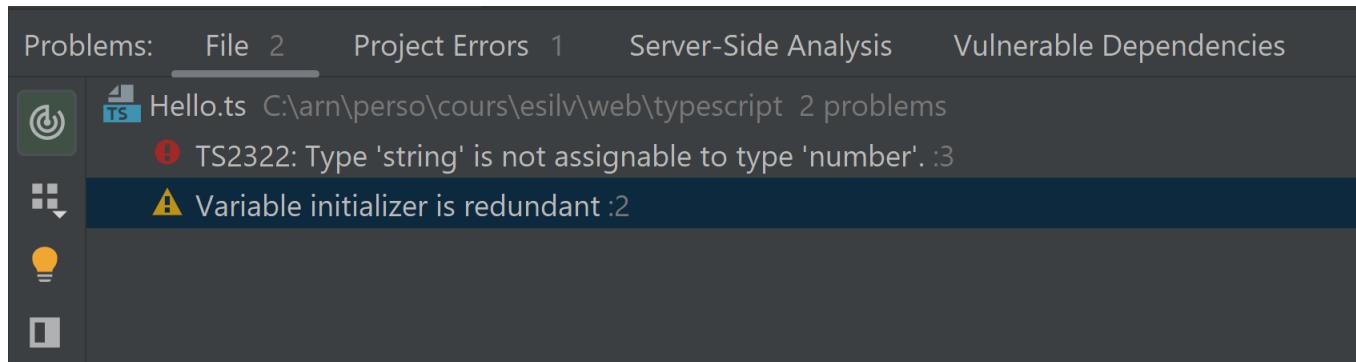
Change JavaScript language version to Flow Alt+Maj+Entrée More actions... Ctrl+1 Alt+Entrée

Compile Time checking ...



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

valid in untyped JavaScript
but not in TypeScript



IntelliJ TypeScript : Recompile on changes

The screenshot shows the 'Languages & Frameworks' configuration window in IntelliJ IDEA. The left sidebar lists various languages and frameworks, with 'TypeScript' currently selected and expanded. The main panel displays settings for the selected language. Under 'TypeScript', the 'Node interpreter' is set to 'Project node (C:\apps\js\nodejs-20.10\node.exe)' and 'Version' is '20.10.0'. The 'TypeScript' dropdown is set to 'Bundled'. In the 'TypeScript' section, several checkboxes are present: 'TypeScript language service' (checked), 'Show project errors' (checked), 'Show suggestions' (checked), and 'Recompile on changes' (checked). A red box highlights the 'Recompile on changes' checkbox. Below these options is a 'Options:' section which is currently empty. At the bottom of the dialog are buttons for '?', 'OK', 'Cancel', and 'Apply'.

Languages & Frameworks > **TypeScript**

Node interpreter: Project node (C:\apps\js\nodejs-20.10\node.exe) 20.10.0

TypeScript: Bundled 5.1.3

TypeScript language service

Show project errors

Show suggestions

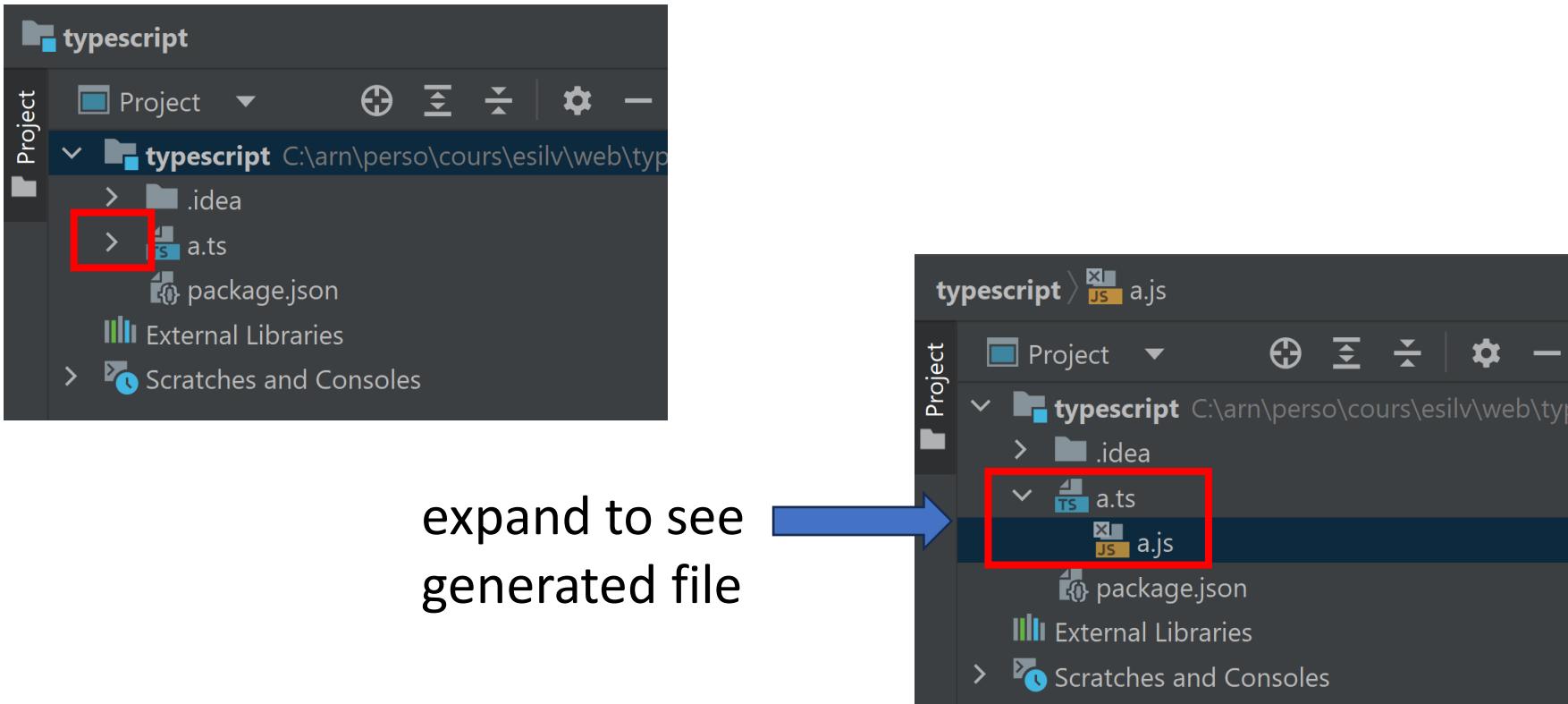
Recompile on changes

Options:

?

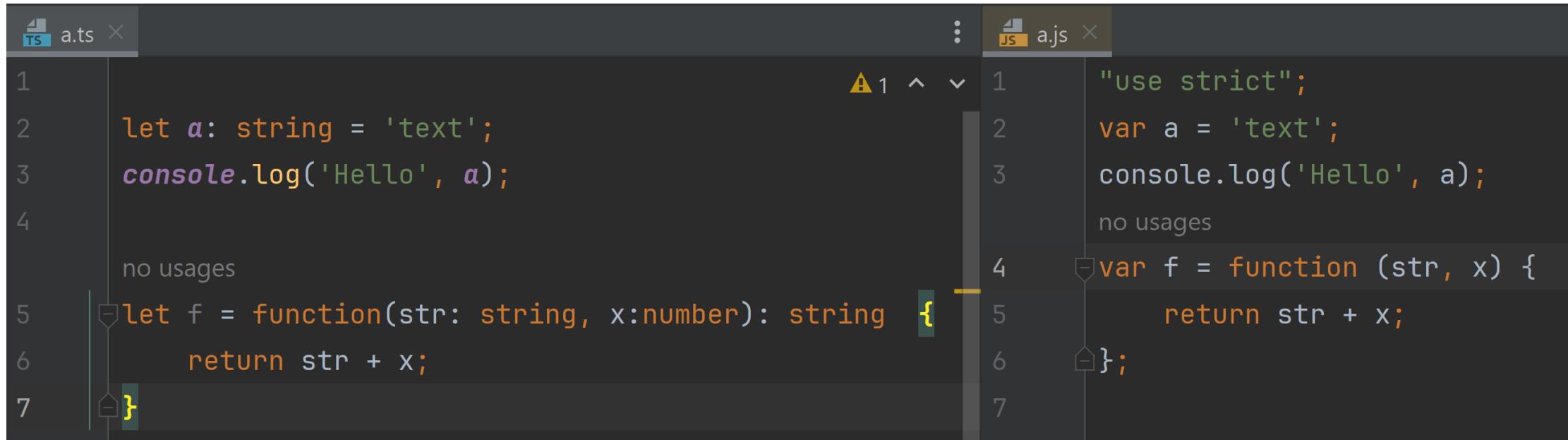
OK Cancel Apply

Js file generated from Ts file



Transpile TypeScript

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



The image shows a code editor with two tabs: 'a.ts' and 'a.js'. The 'a.ts' tab contains TypeScript code, and the 'a.js' tab contains the generated JavaScript code.

a.ts (TypeScript)

```
1 let a: string = 'text';
2 console.log('Hello', a);
3
4 no usages
5 let f = function(str: string, x:number): string {
6     return str + x;
7 }
```

a.js (JavaScript)

```
1 "use strict";
2 var a = 'text';
3 console.log('Hello', a);
4
5 var f = function (str, x) {
6     return str + x;
7 }
```

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.

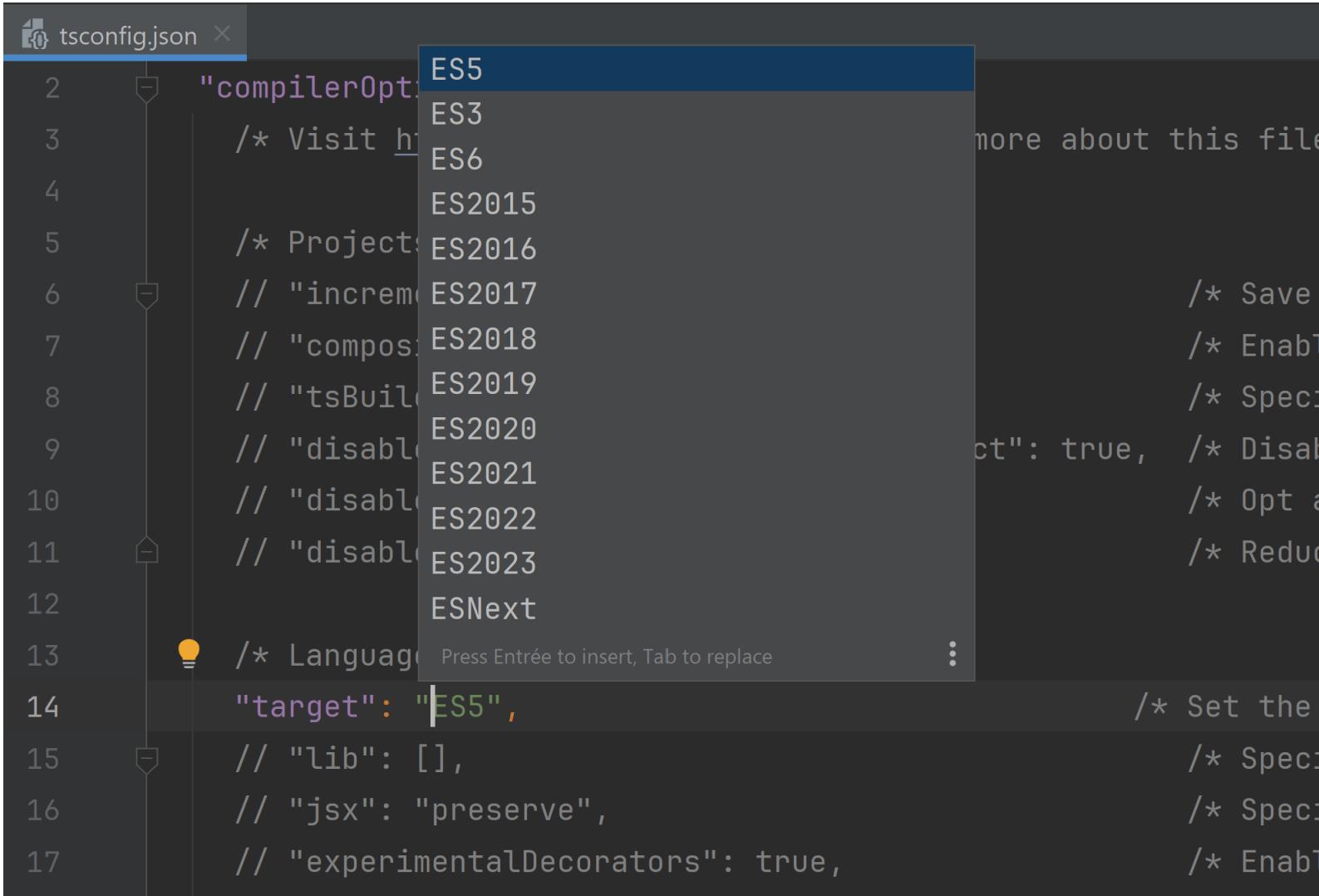
tsc -b
Build a composite project in the working directory.

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



```
tsconfig.json
 2   "compilerOptions": {
 3     /* Visit https://aka.ms/tsc/config to learn about the/compilerOptions in tsconfig.json */
 4     /* Projects */
 5     /* Incremental */
 6     // "incremental": true, /* Save disk space by only emitting changes */
 7     // "composite": true, /* Enable workspace compaction */
 8     // "tsBuildInfoFile": "tsBuildInfoFile.json", /* Specify build info file */
 9     // "disableOptimizingHelpers": true, /* Disable optimization helpers */
10     // "disableSourceMap": true, /* Optimize source maps */
11     // "disableEmitMetadata": true, /* Reduce metadata output */
12     // "disableOpenExternal": true, /* Reduce metadata output */
13     /* Languages */
14     "target": "ES5", /* Set the target ES version */
15     // "lib": [], /* Specify library files to be included */
16     // "jsx": "preserve", /* Specify JSX processing */
17     // "experimentalDecorators": true, /* Enable experimental decorators */
```

The screenshot shows a code editor window with the file 'tsconfig.json' open. The cursor is positioned at the end of the line ' "target": "ES5",'. A dropdown menu is displayed, listing several ECMAScript versions: ES5, ES3, ES6, ES2015, ES2016, ES2017, ES2018, ES2019, ES2020, ES2021, ES2022, ES2023, and ESNext. The option 'ES5' is highlighted with a blue background. The code editor interface includes a status bar at the bottom with the text 'Press Entrée to insert, Tab to replace'.

Example Emulating class with ES5

```
a.ts
1 no usages
2 class Pt {
3     no usages
4     constructor(public x: number, public y: number)
5 }
6 no usages
7 toString(): string {
8     return '(' + this.x + ', ' + this.y + ')';
9 }
10 }
```

```
a.js
1 "use strict";
2 var Pt = /** @class */ (function () {
3     function Pt(x, y) {
4         this.x = x;
5         this.y = y;
6     }
7     Pt.prototype.toString = function () {
8         return '(' + this.x + ', ' + this.y + ')';
9     };
10    return Pt;
11}());
12
```

JavaScript evolve (from TypeScript ESNext)

... example: have class since ES2015, after TS had

The screenshot shows two code editors side-by-side, illustrating the evolution of a class from TypeScript to plain JavaScript.

Left Editor (Pt.ts):

```
1 no usages
2 class Pt {
3     no usages
4     constructor(public x: number, public y: number) {
5         }
6     no usages
7     toString(): string {
8         return '(' + this.x + ', ' + this.y + ')';
9     }
10}
```

Right Editor (Pt.js):

```
1 "use strict";
2 no usages
3 class Pt {
4     no usages
5     constructor(x, y) {
6         this.x = x;
7         this.y = y;
8     }
9     no usages
10    toString() :string {
11        return '(' + this.x + ', ' + this.y + ')';
12    }
13}
```

The code is identical in both files, demonstrating that the TypeScript code is transpiled into plain JavaScript. The visual representation includes syntax highlighting and code folding indicators.

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 Immediately Invoked Anonymous Function

But still weird

JavaScript has weird "var" visibility

"use strict"; // at first line

use Immediately Invoked Anonymous Function

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

use TypeScript

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

A screenshot of a code editor window titled "block-decl.js". The code is as follows:

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

The code uses the "use strict" directive at the top. It contains a block of code enclosed in curly braces. Inside this block, a variable "a" is declared with the type annotation "number" and initialized to 1. A call to "console.log" is made with "a" as the argument. The code editor highlights the variable "a" with a blue background and shows a yellow lightbulb icon, indicating it's a declaration. The output of the code is shown in the "Debugger Console" tab below.

A screenshot of the "Debugger Console" tab from the previous code editor window. The output is:

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

The output shows the string "after decl let, in same block" followed by the value "1", which corresponds to the value of the variable "a" logged from the code above.

JavaScript has weird "var" visibility



The screenshot shows a code editor window with a dark theme. The file is named 'block-decl.js'. The code is as follows:

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

The variable 'a' is highlighted with a blue selection bar. The code editor's interface includes tabs, a status bar at the bottom, and a sidebar on the left.

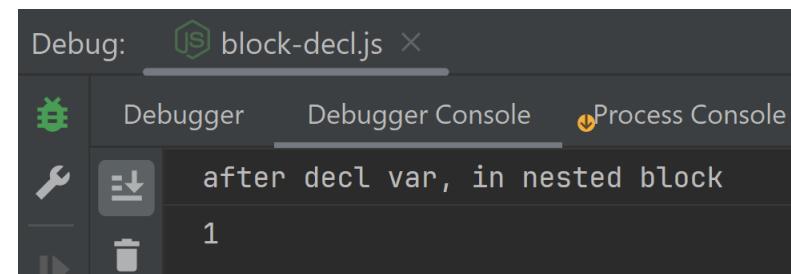
What do you think ??

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

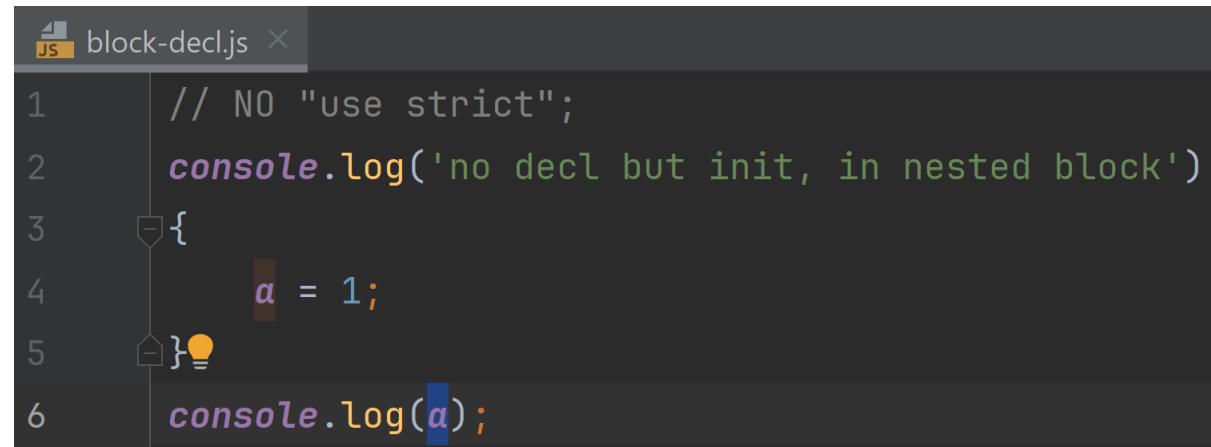
Answer: d/ print 1
the scope of visibility if the enclosing function
(not the enclosing block)

The screenshot shows a code editor window with a dark theme. The file is named 'block-decl.js'. The code contains the following:

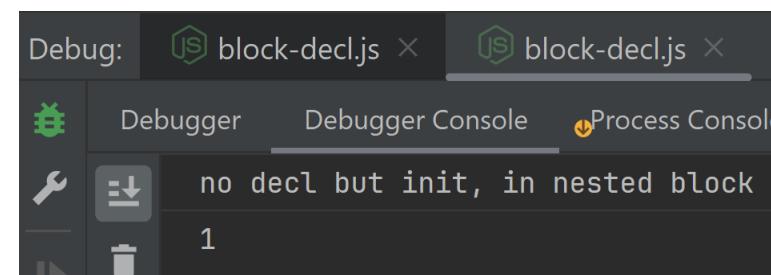
```
1 "use strict";
2 console.log('after decl var, in nested block');
3 {
4     var a : number = 1;
5 }
6 console.log(a);
```



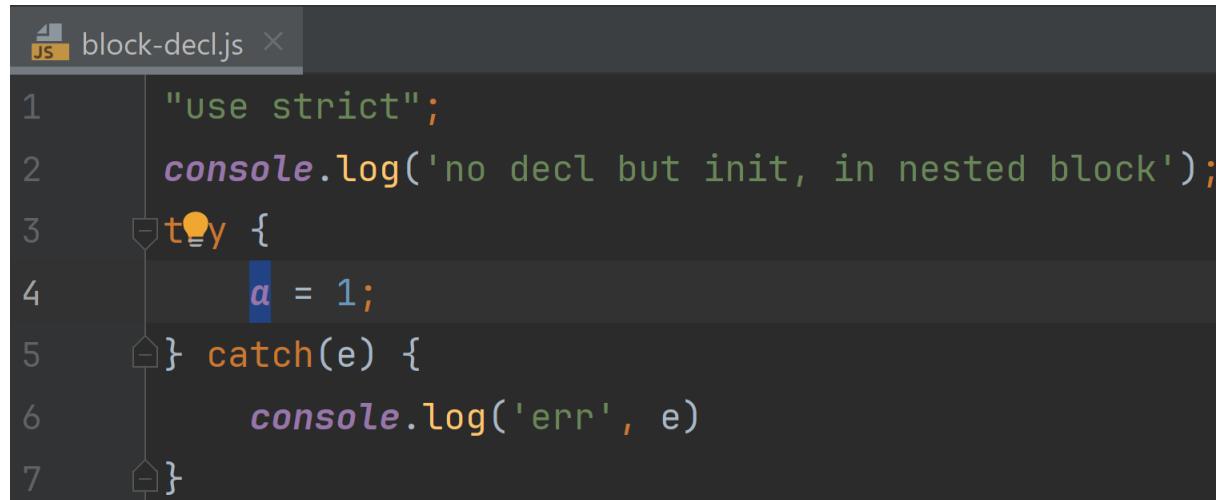
without "use strict" ... NO need to decl "var"



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

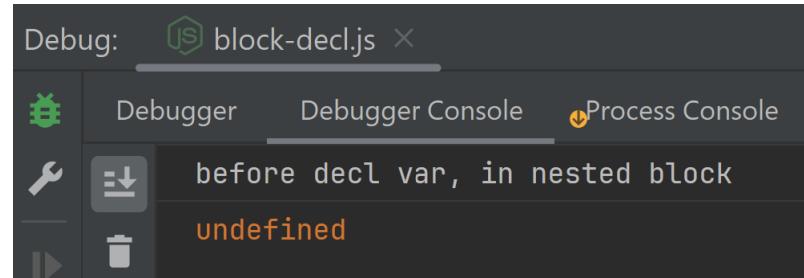
err

> ReferenceError: a is not defined at Object.<anonymous> (C:\arn\perso\cours\etc\block-decl.js:5:1)
(node:internal/modules/cjs/loader:1376:14) at Module._extensions..js (node:internal/modules/extension...
(node:internal/modules/cjs/loader:1207:32) at Module._load (node:internal/modules/loader:
(node:internal/modules/run_main:135:12) at node:internal/main/run_main_module (node:
node:internal/main/run_main_module:28:49", message: "a is not defined")}

even stranger: use before declare in nested block

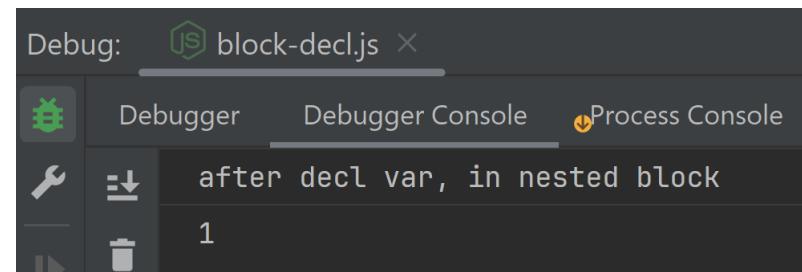
The screenshot shows a code editor window with a dark theme. The file is named 'block-decl.js'. The code contains the following:

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



other strange "var" visibility

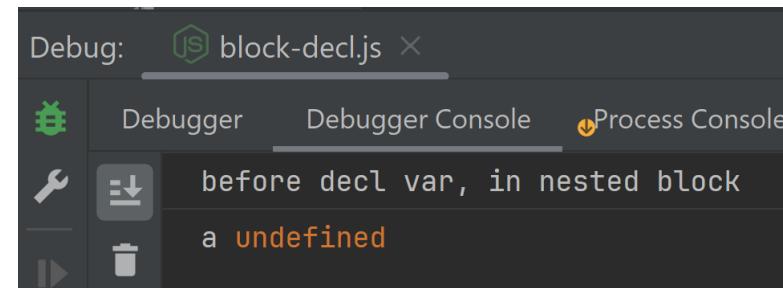
```
1  "use strict";
2  console.log('after decl var, in nested block');
3  {
4      var a : number = 1;
5  }
6  console.log(a);
```



JavaScript has weird "var" visibility

The screenshot shows a code editor window with a dark theme. The file is named 'block-decl.js'. The code contains the following:

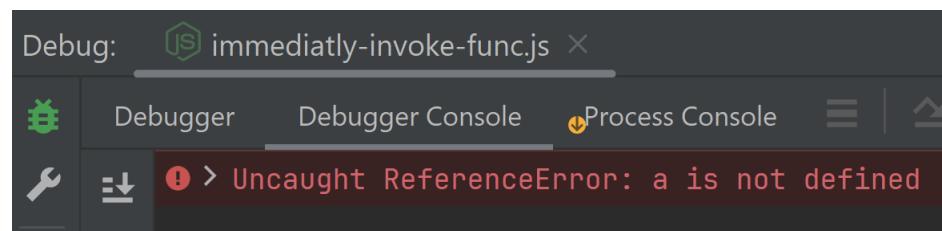
```
1  "use strict";
2  console.log('before decl var, in nested block');
3  console.log(a);
4  {
5      var a : number = 1;
6  }
```



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

work around for scope : "Immediately Invoked Anonymous Function"

```
immediately-invoke-func.js ×
1 "use strict";
2 (function () {
3     var a = 12;
4 })();
5 console.log(a);
```



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

```
JS 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);
```

```
JS 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  }
```

Debug: JS block-decl.js ×

Debugger Debugger Console Process Console

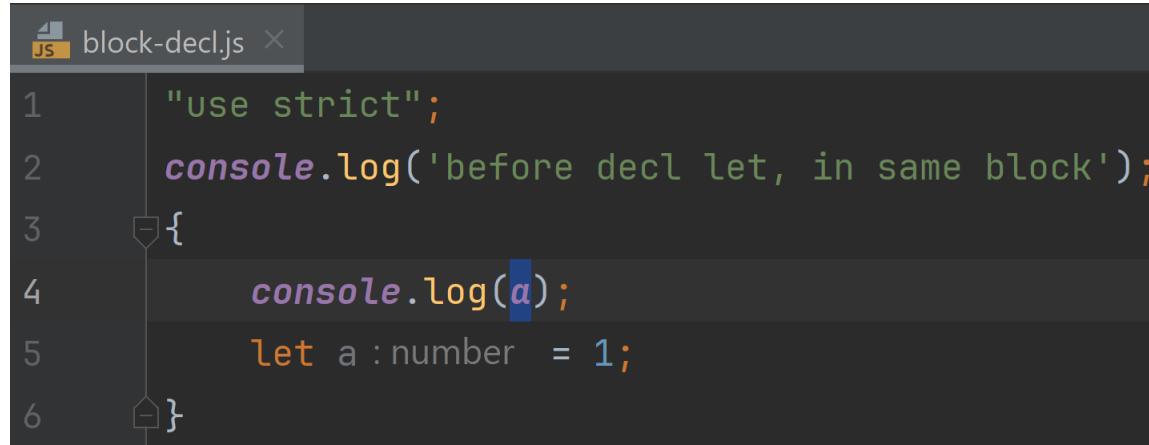
after decl let, in nested block
! > Uncaught ReferenceError: a is not defined

Debug: JS block-decl.js ×

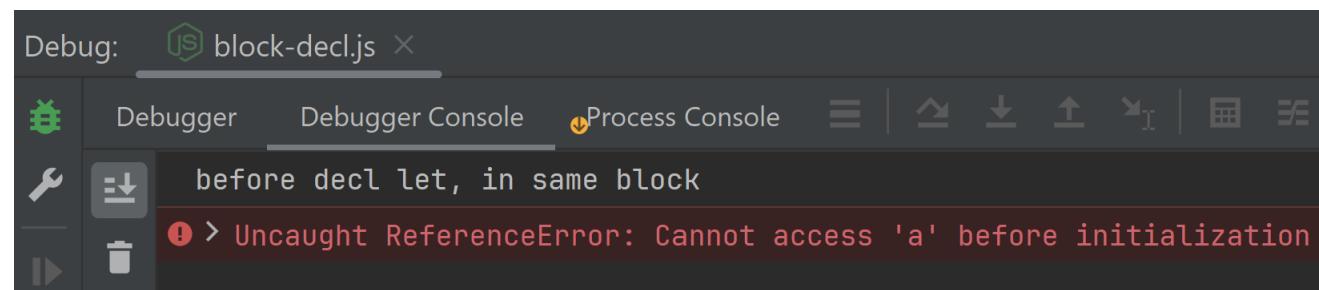
Debugger Debugger Console Process Console

before decl let, in nested block
! > Uncaught ReferenceError: a is not defined

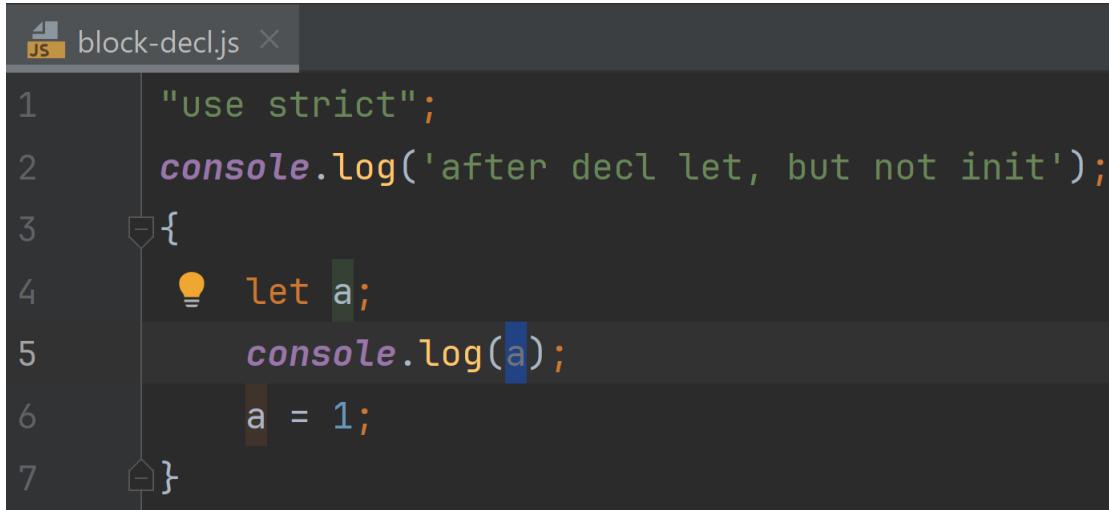
"let" (or "const") in same block



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



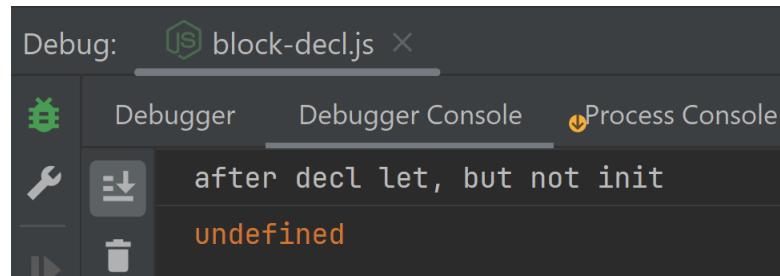
not init (ok in JS !)
but does not compile in TypeScript



The screenshot shows a code editor window titled "block-decl.js". The code is as follows:

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

The code uses a block-scoped variable declaration ("let") without an initial value. The line "console.log(a);" is highlighted with a blue selection bar.



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

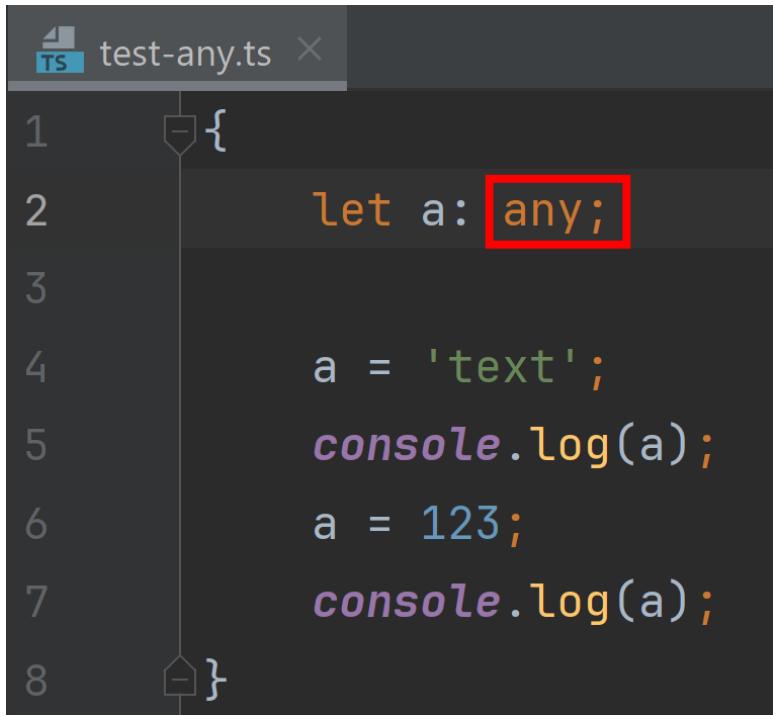
Array

Interfaces : type constraint on existence of field / partial field in Object

" ? " Optional Field Type (for accepting "undefined")

Function, Class, Templates, etc.

any ?



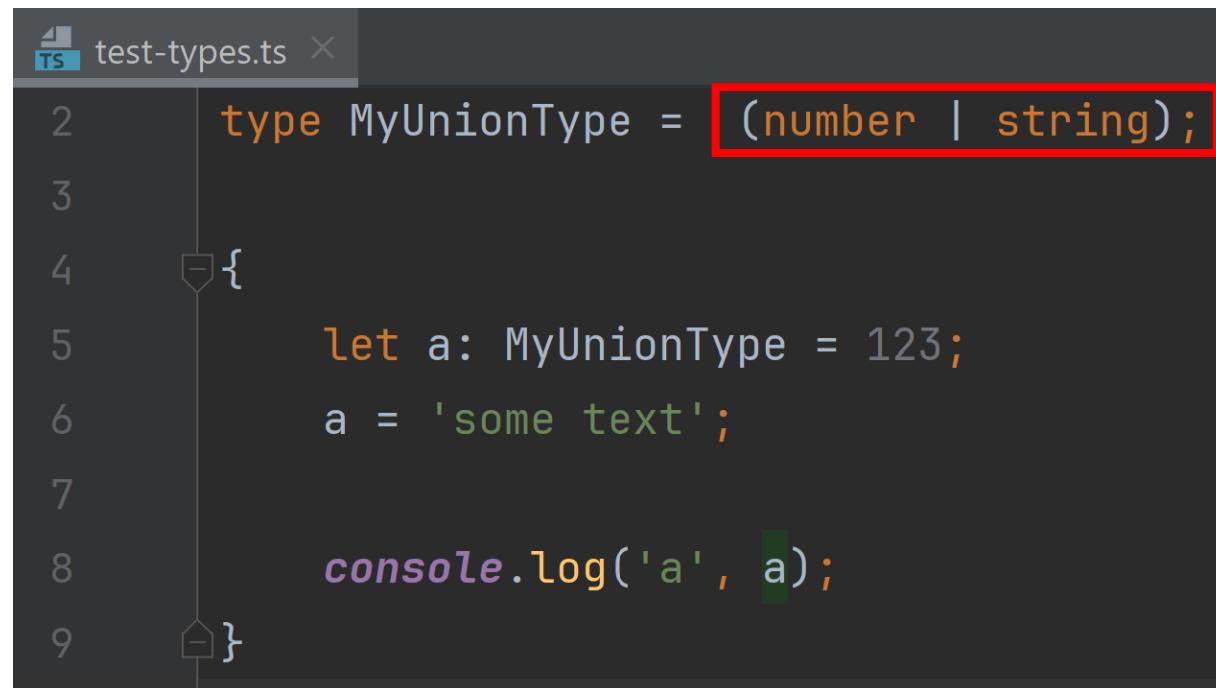
```
1  {
2      let a: any;
3
4      a = 'text';
5      console.log(a);
6      a = 123;
7      console.log(a);
8  }
```

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



The screenshot shows a code editor window with a dark theme. The file is named 'test-types.ts'. The code defines a union type 'MyUnionType' which can be either a 'number' or a 'string'. It then declares a variable 'a' of type 'MyUnionType' and initializes it with both a number and a string. Finally, it logs the value of 'a' to the console.

```
1 // test-types.ts
2 type MyUnionType = number | string;
3
4 {
5     let a: MyUnionType = 123;
6     a = 'some text';
7
8     console.log('a', a);
9 }
```

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 "!!"

The screenshot shows a TypeScript code editor window titled "test-force-defined.ts". The code defines a function "myFunc" that takes a parameter "x" of type "(number | undefined)". Inside the function, two let statements are present:

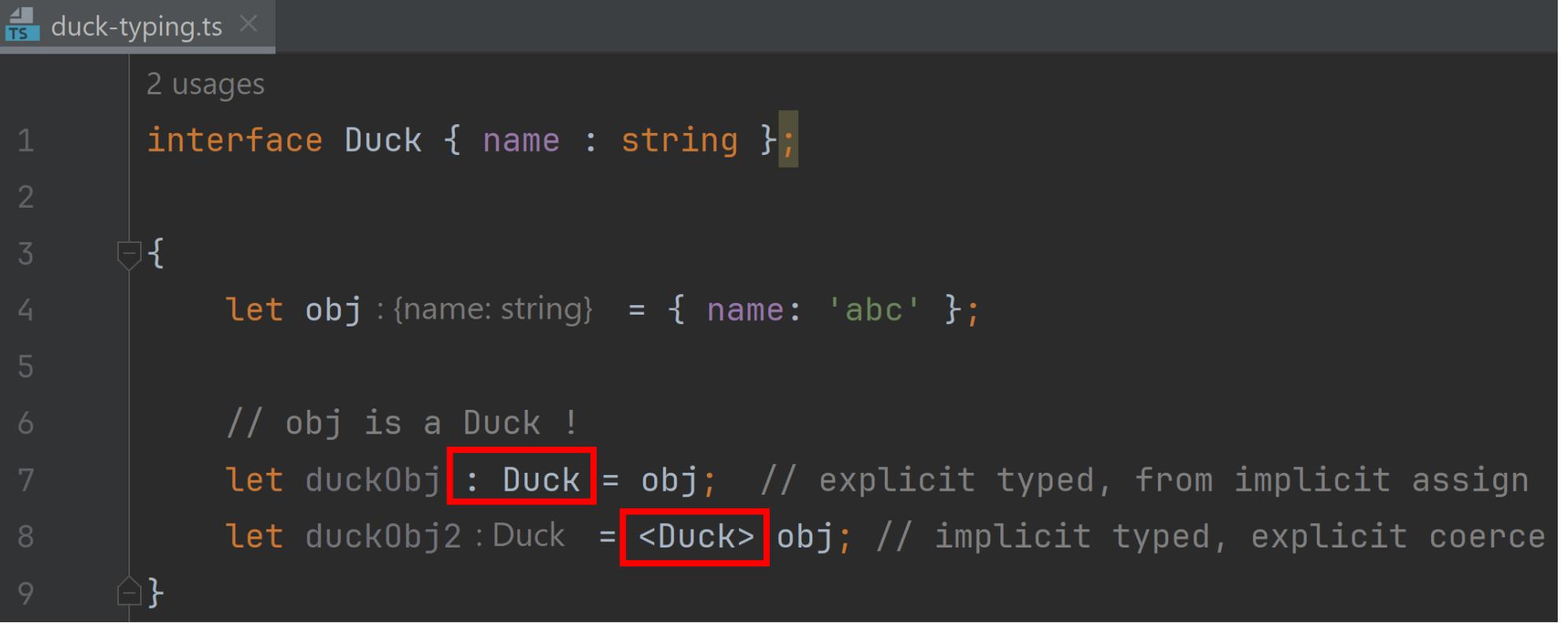
```
function myFunc(x: (number | undefined)) : void {  
    let forceAssign : number = x!!;  
    let errorAssign : number = x;  
}
```

The first assignment, "let forceAssign : number = x!!;", is highlighted with a red box around the "x!!" part. A tooltip appears below the cursor, displaying the error message: "TS2322 Type number | undefined is not assignable to type number". Below this, another message states: "Type undefined is not assignable to type number".

At the bottom of the tooltip, there are several actions: "Remove unused local variable 'errorAssign'" (Alt+Maj+Entrée), "More actions...", and keyboard shortcuts "Ctrl+1 Alt+Entrée".

The status bar at the bottom of the editor shows "test-force-defined.ts" and the "typescript" logo.

Duck Typing



The screenshot shows a code editor window titled "duck-typing.ts". The code defines an interface "Duck" and demonstrates two ways to use it:

```
1  interface Duck { name : string };  
2  
3  {  
4      let obj :{name: string} = { name: 'abc' };  
5  
6      // obj is a Duck !  
7      let duckObj : Duck = obj; // explicit typed, from implicit assign  
8      let duckObj2 :Duck = <Duck> obj; // implicit typed, explicit coerce  
9  }
```

The code editor highlights the word "Duck" in orange. The assignment in line 7 and the type annotation in line 8 are both enclosed in red boxes.

Structural Interface Typing, Combine Partial

```
test-interface.ts
1  interface Ix { x: number; }
2
3  interface Iy { y: number; }
4
5  type Ixy = Ix & Iy;
6
7  type Pt = { x: number; y: number }; // equivalent to Ix & Iy
8
9
10 let objX = { x: 1 }; // type inferred = { x:number }
11
12 let objX2 : Ix = objX; // implicit type "cast" ok
13
14 let objY = { y: 1 };
15
16 let obj: Pt = { ...objX, ...objY };
17
18 console.log(obj);
19
20 }
```

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

Type Checking "missing property" in Interface

The screenshot shows a code editor window for a file named `test-missing-property.ts`. The code defines an interface `Pt` with properties `x` and `y`, and creates a variable `obj` of type `Pt` with only the `x` property assigned.

```
1 usage
2
3 interface Pt { x: number, y: number; }
4 let obj: Pt = { x: 12 };
5 }
```

A red box highlights the error message in the status bar: `TS2741: Property y is missing in type { x: number; } but required in type Pt`. Below the message, it says `Pt.ts(4, 35): y is declared here.`

Actions available in the status bar include:

- Remove unused local variable 'obj' Alt+Maj+Entrée
- More actions... Ctrl+1 Alt+Entrée

At the bottom of the status bar, there is a file icon for `typescript`.

TypeScript...

TypeScript is AWESOME !

allow to write very concise functional / json structured code

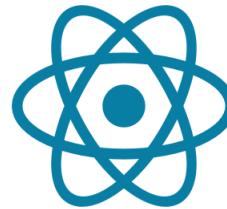
What about JSX / TSX ?

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>); }

← → ⌂ react.dev/learn



v18.3.1

Search

Ctrl K

GET STARTED

Quick Start

Tutorial: Tic-Tac-Toe

Thinking in React

Installation

LEARN REACT

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

React components are JavaScript functions that return markup:

```
function MyButton() {
  return (
    <button>I'm a button</button>
  );
}
```

Now that you've declared `MyButton`, you can nest it into another component:

```
export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton />
    </div>
  );
}
```

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

The screenshot shows the official React documentation website. At the top, there's a navigation bar with the React logo, version v18.3.1, a search bar, and links for 'Ctrl K', 'Learn', and 'Reference'. On the left, a sidebar titled 'GET STARTED' has a 'Quick Start' section currently selected, which includes links to 'Tutorial: Tic-Tac-Toe' and 'Thinking in React'. Below this are sections for 'Installation', 'LEARN REACT', 'Describing the UI', 'Adding Interactivity', 'Managing State', and 'Escape Hatches'. The main content area is titled 'Displaying data' and contains text explaining JSX templating. It shows a code snippet:return (
 <h1>
 {user.name}
 </h1>
);A yellow highlight box surrounds the part where the user variable is used in the JSX template. Below this, another text block explains how to escape into JavaScript from attributes:

You can also “escape into JavaScript” from JSX attributes, but you have to use curly braces instead of quotes. For example, `className="avatar"` passes the `"avatar"` string as the CSS class, but `src={user.imageUrl}` reads the JavaScript `user.imageUrl` variable value, and then passes that value as the `src` attribute:

```
return (  
  <img  
    className="avatar"  
    src={user.imageUrl}  
  />
```

A second yellow highlight box surrounds the `src={user.imageUrl}` part of the code. At the bottom left, there's a feedback section asking 'Is this page useful?' with thumbs up and down icons.

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

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

The screenshot shows the React documentation website at version v18.3.1. The main heading is "Function instead of 'Class' object instances ! where to store field state?". Below it, the sub-section title is "First, import `useState` from React:". A code snippet shows the import statement:

```
import { useState } from 'react';
```

Below this, the text says "Now you can declare a *state variable* inside your component:" followed by a code snippet:

```
function MyButton() {
  const [count, setCount] = useState(0);
  // ...
```

At the bottom, there is explanatory text: "You'll get two things from `useState`: the current state (`count`), and the function that lets you update it (`setCount`). You can give them any names, but the convention is to write `[something, setSomething]`.

The left sidebar contains navigation links for "GET STARTED", "Quick Start" (which is currently selected), "Tutorial: Tic-Tac-Toe", "Thinking in React", "Installation", "LEARN REACT", "Describing the UI", "Adding Interactivity", "Managing State", and "Escape Hatches". The "Learn" tab is also visible in the top right.

And How to pass input values from parent to child?

GET STARTED

Quick Start

Tutorial: Tic-Tac-Toe

Thinking in React

Installation

LEARN REACT

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

```
<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>

The screenshot shows the Babel.js documentation website. The top navigation bar includes the BABEL logo, a search icon, and links for Docs and Setup. The left sidebar lists various Babel components: Guides, Config Reference, Presets (with @babel/preset-env and @babel/preset-react expanded), and sections for transform-react-jsx, @babel/preset-typescript, @babel/preset-flow, Misc, Integration Packages, Tooling Packages, and Helper Packages. The @babel/preset-react section is currently selected, as indicated by a highlighted background. In the main content area, under the 'Out' section, there are two tabs: JavaScript and JSX. The JavaScript tab displays the translated code from JSX to JavaScript using the `_jsxs` and `_jsx` functions. The JSX tab shows the original JSX code.

In

```
const profile = (
  <div>
    
    <h3>{[user.firstName, user.lastName].join(" ")}</h3>
  </div>
);
```

Out

JavaScript

```
import { jsx as _jsx } from "react/jsx-runtime";
import { jsxs as _jsxs } from "react/jsx-runtime";

const profile = _jsxs("div", {
  children: [
    _jsx("img", {
      src: "avatar.png",
      className: "profile",
    }),
    _jsx("h3", {
      children: [user.firstName, user.lastName].join(" "),
    }),
  ],
});
```

JSX

```
<div>
  
  <h3>{[user.firstName, user.lastName].join(" ")}</h3>
</div>
```

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

Fortunately, 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]

GET STARTED

Quick Start >

Installation ▾

Start a New React Project

Add React to an Existing Project

Editor Setup

Using TypeScript

React Developer Tools

React Compiler 

LEARN REACT

Describing the UI >

Adding Interactivity >

Managing State >

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.

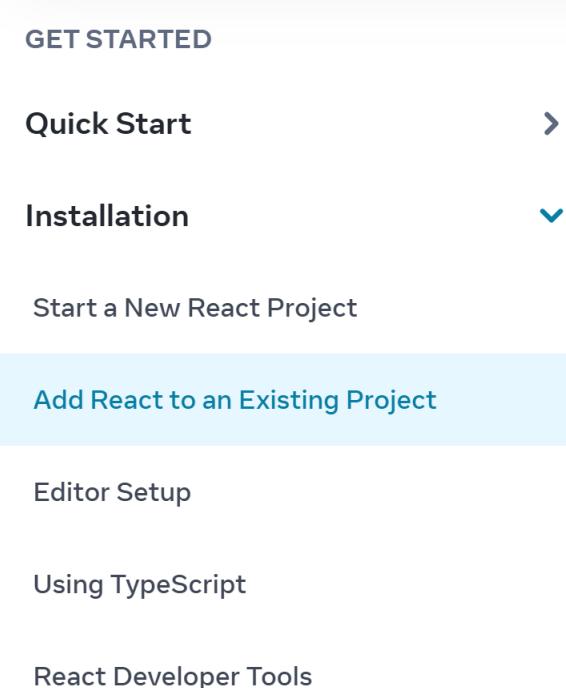
To check whether your setup works, run this command in your project folder:

Terminal

 Copy

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

Integrating React in a Web Page [2/2]



Step 2: Render React components anywhere on the page

In the previous step, you put this code at the top of your main file:

```
import { createRoot } from 'react-dom/client';

// Clear the existing HTML content
document.body.innerHTML = '<div id="app"></div>';

// Render your React component instead
const root = createRoot(document.getElementById('app'));
root.render(<h1>Hello, world</h1>);
```

JS -> JSX, TypeScript -> TSX

GET STARTED

Quick Start

Installation

Start a New React Project

Add React to an Existing Project

Editor Setup

Using TypeScript

React Developer Tools

React Compiler 

LEARN REACT

Describing the UI

Adding Interactivity

Managing State

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
- ..