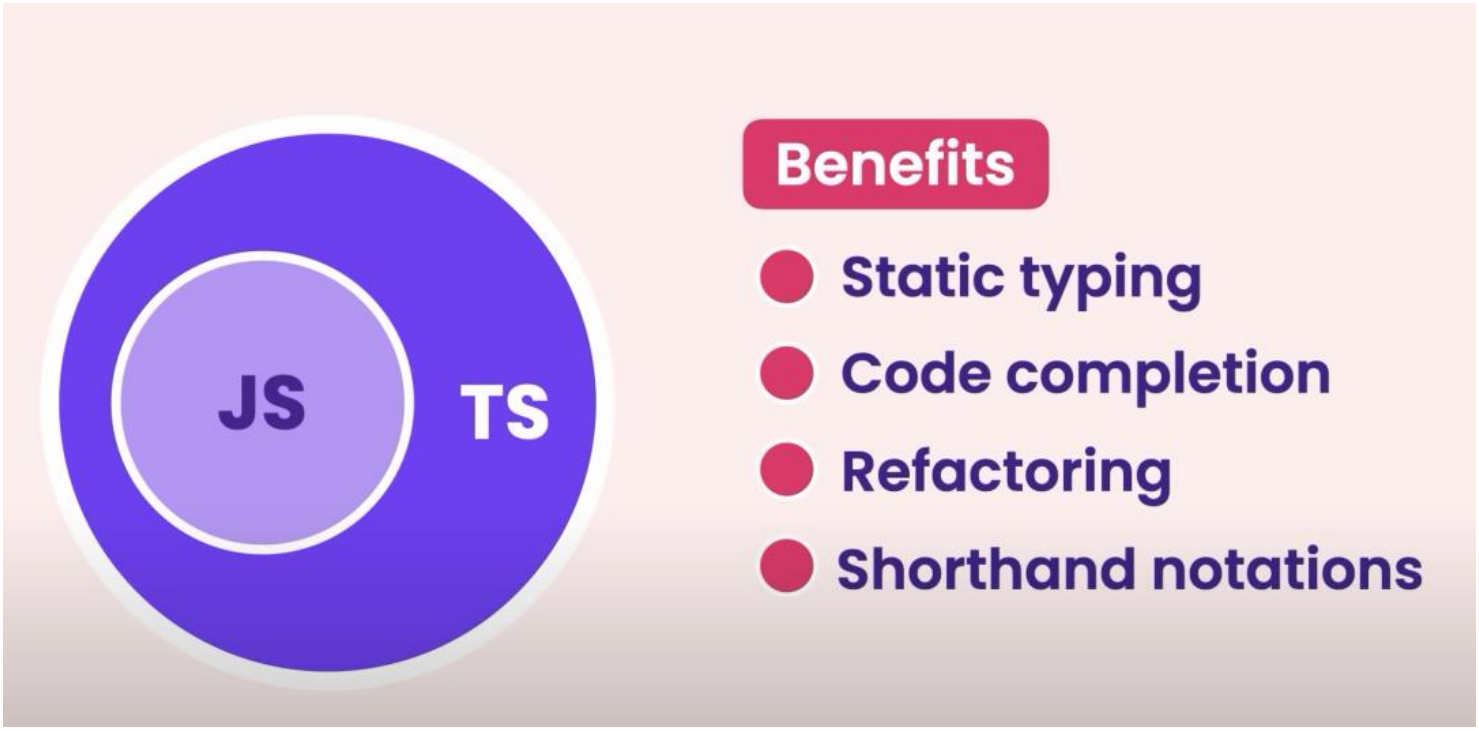
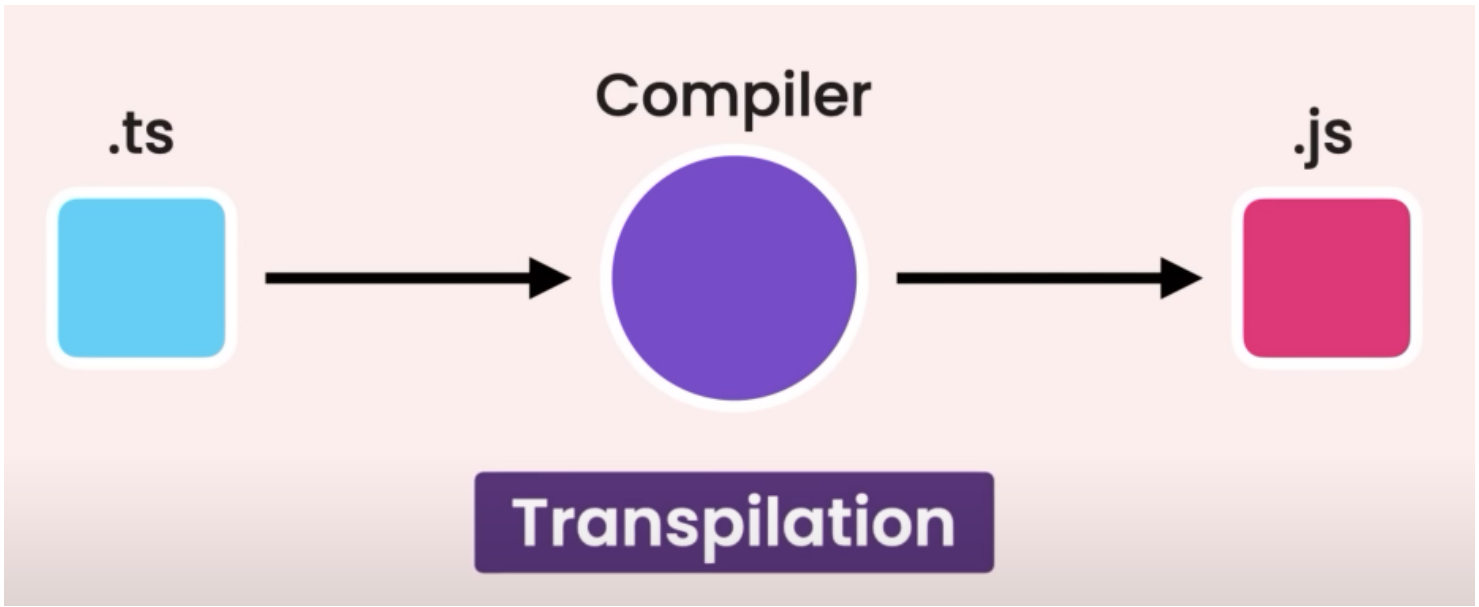
Typescript

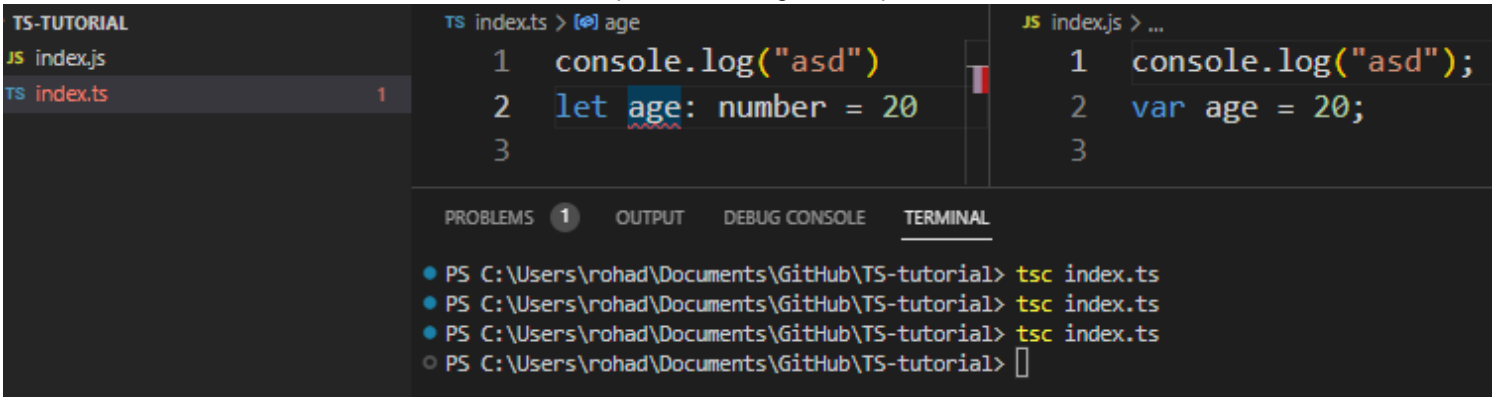


Statically typed languages like c++, you cannot reassign the type of the variable. So let x = 10 cannot be reassigned as let x = “string”. In dynamically typed languages, this gives the dev a lot flexibility, but when we not necessarily pick out these bugs until we compile. Typescript prevent us to have this issue as it always has to compile to JS code. As browser doesn`t understand TS it always have to compile to JS. It also means that we have to be more disciplined coder.




```
ritual.joint@ritual-joint MINGW64 /
$ cd /c/Users/rohad/Documents/GitHub

ritual.joint@ritual-joint MINGW64 ~/Documents/GitHub
$ cd TS-tutorial

ritual.joint@ritual-joint MINGW64 ~/Documents/GitHub/TS-tutorial
$ npm i -g typescript

added 1 package in 6s
npm notice
npm notice New minor version of npm available! 9.3.1 -> 9.8.1
npm notice Changelog: <https://github.com/npm/cli/releases/tag/v9.8.1>
npm notice Run `npm install -g npm@9.8.1` to update!
npm notice

ritual.joint@ritual-joint MINGW64 ~/Documents/GitHub/TS-tutorial
$ tsc -v
Version 5.1.6
```

compile ts to js example



tsc –init creates a configuration file where its commented out what can be used. Uncommenting a few neeeded such:

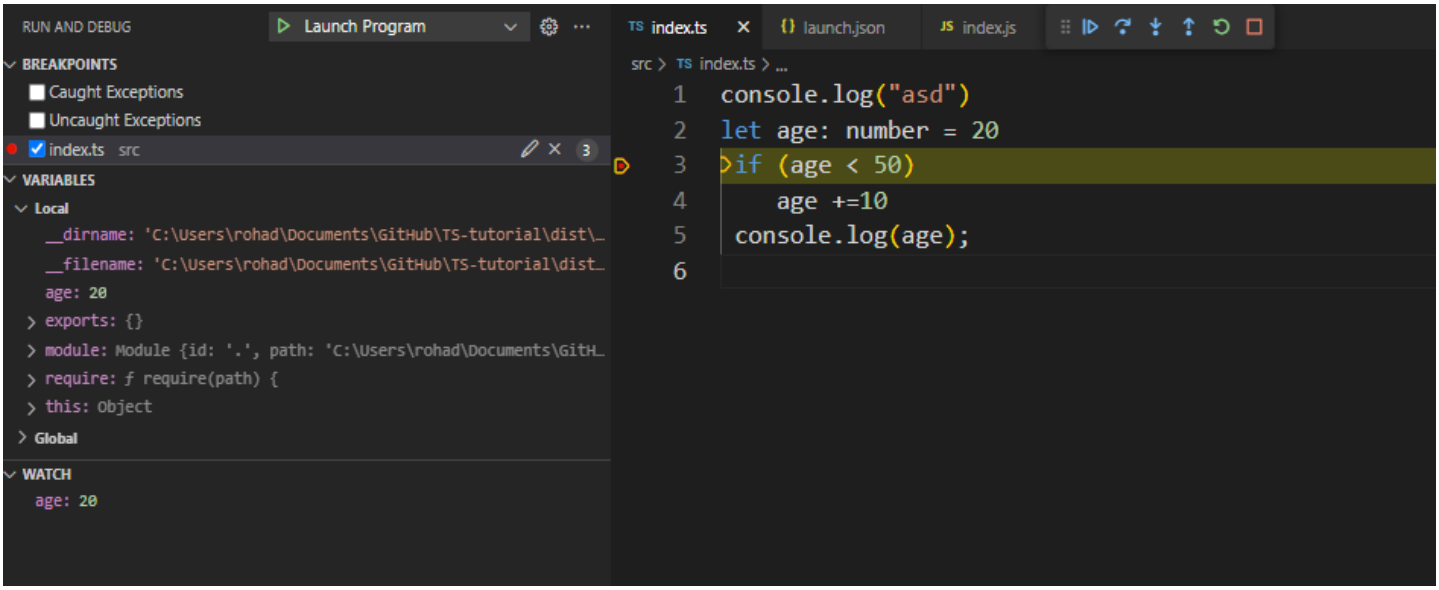
- **"rootDir": "./",** ← in the name the root of the folder
- **"outDir": "./dist",** ← the folder where compiled JS file will be
- **"noEmitOnError": true,** ← not creating/compile JS file on errors
- **"removeComments": true,** ← removing comments TS -> JS so its gonna be shorter
- **"sourceMap": true,** ← debugging, creates index.js.map

Debugging

VS-code -> ○ Insert breakpoint (execution will stop at this point and we can debug line by line) -> Run and Debug -> Create a -> node

It creates a launch.json file with some configurations we add:
"preLaunchTask": "tsc: build - tsconfig.json", ← meaning: use TS compiler to build application with this configuration file

Lunch program F5 – shortcut
We can also watch variables

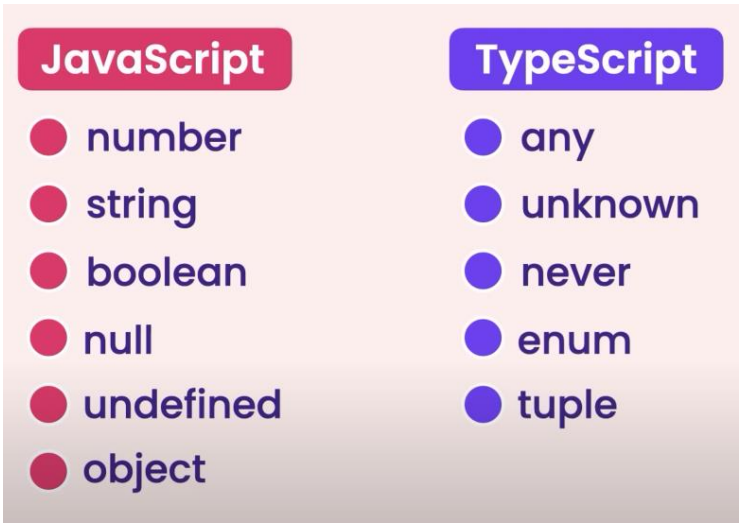


TS variables

Once we declare something to be a number:
let sales: number = 123_456_789; even if we change it to let = 123_456_789 it will remain as number.

```
let sales: number
let sales = 123_456_789;
```

In TS there are more types of variables available.



If we don`t initialise a variable:
let level; it will declare it any. any can be number or later declared to be a string, but its against the whole idea of TS so try not to use it.

In JS arrays can be different types:
let numbers = [1, 2, '3'] <- this is totally valid as in JS arrays are dynamic. We need to apply a type annotation in TS:
let numbers: number[] = [1, 2, '3'] this will now bring up an error in TS. If we would have started with let numbers = [1, 2, 3] TS would know its only numbers, but if the array is empty let numbers = [] it will declare it any.

Tuples fixed length array

let user: [number, string] = [1, "Andras"] its an array with only two element (not more nor less!), it also will keep on eye the types of the elements so if we make ["1", “Andras”] it will flag it that the first element should be a number.(Be aware if we compile Tuples to JS it will be a simple array and won`t complain if you would us push() method).

Enum list of related constants

```
const small = 1;
const medium = 2;
const large = 3;
```

//use PascalCase
enum Size { Small, Medium, Large}; || enum Size { Small = 1, Medium = 2, Large = 3 } <- Be aware if we don’t declare { Small, Medium, Large } it would automatically would be 0, 1, 2 . If you set Small = 1 and don`t declare the other two it would assign Medium = 2 and Large = 3.

It will compile to this in JS

```
//use PascalCase
enum Size { Small = 1, Medium, Large };
let mySize: Size = Size.Medium;
console.log(mySize);
```

```
14  var Size;
15  (function (Size) {
16      Size[Size["Small"] = 1] = "Small";
17      Size[Size["Medium"] = 2] = "Medium";
18      Size[Size["Large"] = 3] = "Large";
19  })(Size || (Size = {}));
20  ;
21  let mySize = Size.Medium;
22  console.log(mySize);
```

If we use: const enum Size { Small, Medium, Large}; in JS will create a more optimised code.

```
let mySize = 2;
console.log(mySize);
//# sourceMappingURL=index.js.map
```

Functions

Although, is correct, but it won`t flag up issues.

```
function calculateTax(anincome: number) {
     return "3"
}
```

Always declare and **annotate** your functions!

```
function calculateTax(anincome: number): number {
     return 3
}
```

```
function calculateTax(anincome: number): number {
     return "Andras"
}
```

Examples, Annotate, non-annotate, annotate as void

```
//anotated will flag it no return or wrong type
function calculateTax0(anincome: number): number {
   return 3;
}

//not annotated, didnt flag empty return
function calculateTax1(anincome: number) {

}

//anotated as void, didnt flag it empty return
function calculateTax2(anincome: number): void {
}
```

Empty parameters

Here is anincome is declared, but never been used. In Ts config files we can set: **"noUnusedParameters": true**, <- it will flag it for us.

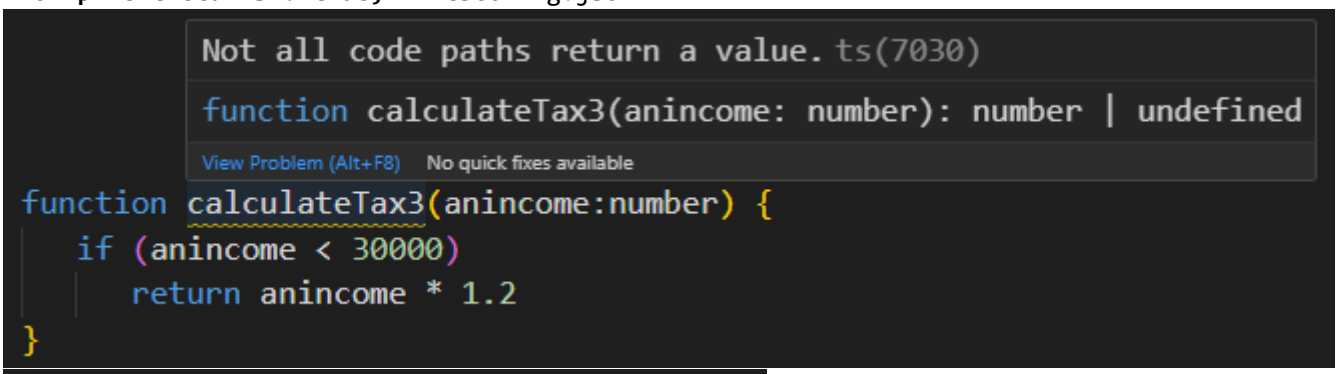
```
'anincome' is declared but its value is never read. ts(6133)
(parameter) anincome: number
View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
//functions
function calculateTax(anincome: number) {
     return "3"
}
```

Undefined returns/noImplicitReturns

if anincome would be more than 30000 it would return undifined break the app.

```
function calculateTax3(anincome:number) {
   if (anincome < 30000)
      return anincome * 1.2
}
```

To overcome this we can turn on
"noImplicitReturns": true, in tsconfig.json




```
function calculateTax3(anincome:number) {
   if (anincome < 30000)
      return anincome * 1.2;
   return anincome * 1.3;
}
```

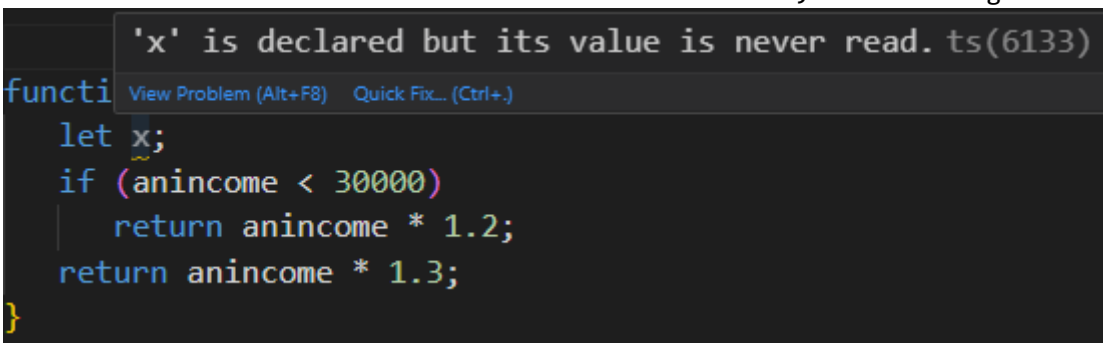
It was an example, but as you can see the returns **WEREN`T ANNOTATED**

No unused vars/noUnusedLocals

We have unused let here, but no error msg.

```
function calculateTax3(anincome:number): number {
   let x;
   if (anincome < 30000)
      return anincome * 1.2;
   return anincome * 1.3;
}
```

To overcome this we can turn on **"noUnusedLocals": true,** not its flag it to us



Arguments

This is valid in TS and JS

```
function calculateTax4(anincome:number, taxYear: number): number {
    if (taxYear < 2022)
        return anincome * 1.2;
    return anincome * 1.3;
}

calculateTax4(10000, 2022)
```

JS would accept this, doesn`t care how many arguments supplied, but TS is strict on this! It wont accept more, it wont accept less.

```
function calculateTax4(anincome:number, taxYear: number): number {
    if (taxYear < 2022)
        return anincome * 1.2;
    return anincome * 1.3;
}

calculateTax4(10000, 2022, 4)
```

Expected 2 arguments, but got 3. ts(2554)

View Problem (Alt+F8) No quick fixes available

But we can make it optional to have the second argument

```
function calculateTax5(anincome:number, taxYear?: number): number {
    if ((taxYear || 2022) < 2022)
        return anincome * 1.2;
    return anincome * 1.3;
}

calculateTax5(10000)
```

But the better way to do this is to declaire an og value to it and if we call the function it will get overwritten.

```
function calculateTax6(anincome:number, taxYear = 2022): number {
    if (taxYear < 2022)
        return anincome * 1.2;
    return anincome * 1.3;
}

calculateTax6(10000, 2023)
```

Objects

Objects are dynamic in JS. We can add properties later on. In TS this is not the case

```
let employee = { id: 1};
employee.name = "Andras"
```

```
//objects  Property 'name' does not exist on type '{ id: number; }'. ts(2339)
let emplo  View Problem (Alt+F8)   No quick fixes available
employee.name = "Andras"
```

We can annotate our object as such, but as you can see we would still have an error as employee exoects to have a name and initally it isn`t declared. We can make it optional as we did with the functions.

```
let employee: {
    id: number,
    name: string
} = { id: 1};
employee.name = "Andras"
```

```
let employee: {
    id: number,
    name: string
} = { id: 1, name: ""};
employee.name = "Andras"
```

```
let employee: {
    id: number,
    name?: string
} = { id: 1};
employee.name = "Andras"
```

Read only

TS complier prevent modifyng value of this propety

```
let employee: {
  readonly id: number,
    name: string
} = { id: 1, name: ""};
employee.id= 2
```

Type aliases

As you can see this implimentation is pretty shit. You would need to repeat the code (DRY principal, don’t repeat yourself), also some employess have parameters that others don’t

```
let employee2: {
  readonly id: number;
  name: string;
  retire: (date: Date) => void;
} = { id: 1, name: "andras", retire: (date: Date) => console.log(date) };
```

Like this our code will be reusable, we wont need to define the parameters again

```
type Employee3 = {
  readonly id: number;
  name: string;
  retire: (date: Date) => void;
};

let employee3: Employee3 = {
  id: 1,
  name: "andras",
  retire: (date: Date) => console.log(date),
};
```

Union types & narrowing

Here input can be number or string

```
function kgToLbs(weight: number | string): number {
  if (typeof weight === "number") return weight * 2.2;
  else return parseInt(weight) * 2.2;
}
kgToLbs(10);
kgToLbs("10kg");
```

Intersection types

UIWidget is both type at the same time

```
type Draggable = {
  drag: () => void
}

type Resizeble = {
  resize: () => void
}

type UIWidget = Draggable & Resizeble;
```

Literal types, exact

Limit values of variable. For example if we annotate let quantity = number; it will accept any number, but what if we only want this to accept 50 or 100?

```
let quantity: 50 | 100;
```

Or using type alias

```
type Quantity = 50 | 100;
let quantity: Quantity = 50;
```

Nullable types

This is valid JS code, but program will crash. Error msg comes from **"strictNullChecks": true,**

```
function greet(name: string){
   console.log(name.toUpperCase);

}
greet(null)
```

Using the union operator

```
function greet(name: string | null){
   if (name)
   console.log(name.toUpperCase);
   else
   console.log("Hola");
}
greet(null)
```

Optional chaining

Optional propety access operator

Cutomer might be null or undifined

```
type Customer = {
  birthday: Date
};

function getCustomer(id: number): Customer | null | undefined {
   return id === 0 ? null : { birthday: new Date() };
}

let customer = getCustomer(0);
if (customer !== null && customer !== undefined)
console.log(customer.birthday);
```

Using customer? We don’t need to repeat the if statement

```
function getCustomer(id: number): Customer | null | undefined {
   return id === 0 ? null : { birthday: new Date() };
}

let customer = getCustomer(0);
console.log(customer?.birthday);
```

But if we need further adjustments, we need to add an other ? as customer.birthday might be null or undefined we can`t run .getFullYear on it (if we have a customer, if customer has a birthday)

```
type Customer = {
  birthday: Date
};

function getCustomer(id: number): Customer | null | undefined {
   return id === 0 ? null : { birthday: new Date() };
}

let customer = getCustomer(0);
console.log(customer?.birthday?.getFullYear);
```

Optional element access operator

If we have and array and we try to accces the first element, it might be null

```
let people = ["sarah", "andras"]
people[3]
```

we need to check it

```
let people = ["sarah", "andras"]
if (people !== null && people !== undefined)
people[3]
```

Or to make it simple

```
let people = ["sarah", "andras"]
people?.[3]
```

Optional call operator

This will only be executed if log is referencing an actual function

```
let log: any = null
log?.("a")
```