

```
{() => fs}
```



b First steps with TypeScript

After the brief introduction to the main principles of TypeScript, we are now ready to start our journey toward becoming FullStack TypeScript developers. Rather than giving you a thorough introduction to all aspects of TypeScript, we will focus in this part on the most common issues that arise when developing Express backends or React frontends with TypeScript. In addition to language features, we will also have a strong emphasis on tooling.

Setting things up

Install TypeScript support to your editor of choice. Visual Studio Code works natively with TypeScript.

As mentioned earlier, TypeScript code is not executable by itself. It has to be first compiled into executable JavaScript. When TypeScript is compiled into JavaScript, the code becomes subject to type erasure. This means that type annotations, interfaces, type aliases, and other type system constructs are removed and the result is pure ready-to-run JavaScript.

In a production environment, the need for compilation often means that you have to set up a "build step." During the build step, all TypeScript code is compiled into JavaScript in a separate folder, and the production environment then runs the code from that folder. In a development environment, it is often easier to make use of real-time compilation and auto-reloading so one can see the resulting changes more quickly.

Let's start writing our first TypeScript app. To keep things simple, let's start by using the npm package ts-node. It compiles and executes the specified TypeScript file immediately so that there is no need for a separate compilation step.

You can install both `ts-node` and the official `typescript` package globally by running:

```
npm install --location=global ts-node typescript
```

[copy](#)

If you can't or don't want to install global packages, you can create an npm project which has the required dependencies and run your scripts in it. We will also take this approach.

As we recall from [part 3](#), an npm project is set by running the command `npm init` in an empty directory. Then we can install the dependencies by running

```
npm install --save-dev ts-node typescript
```

[copy](#)

and setting up `scripts` within the `package.json`:

```
{
  // ..
  "scripts": {
    "ts-node": "ts-node"
  },
  // ..
}
```

[copy](#)

You can now use `ts-node` within this directory by running `npm run ts-node`. Note that if you are using `ts-node` through `package.json`, command-line arguments that include short or long-form options for the `npm run script` need to be prefixed with `--`. So if you want to run `file.ts` with `ts-node` and options `-s` and `--someoption`, the whole command is:

```
npm run ts-node file.ts -- -s --someoption
```

[copy](#)

It is worth mentioning that TypeScript also provides an online playground, where you can quickly try out TypeScript code and instantly see the resulting JavaScript and possible compilation errors. You can access TypeScript's official playground [here](#).

NB: The playground might contain different `tsconfig` rules (which will be introduced later) than your local environment, which is why you might see different warnings there compared to your local environment. The playground's `tsconfig` is modifiable through the config dropdown menu.

A note about the coding style

JavaScript is a quite relaxed language in itself, and things can often be done in multiple different ways. For example, we have named vs anonymous functions, using `const` and `let` or `var`, and the optional use of `semicolons`. This part of the course differs from the rest by using semicolons. It is not a TypeScript-specific pattern but a general coding style decision taken when creating any kind of JavaScript project.

Whether to use them or not is usually in the hands of the programmer, but since it is expected to adapt one's coding habits to the existing codebase, you are expected to use semicolons and adjust to the coding style in the exercises for this part. This part has some other coding style differences compared to the rest of the course as well, e.g. in the directory naming conventions.

Let us add a configuration file `tsconfig.json` to the project with the following content:

```
{
  "compilerOptions": {
    "noImplicitAny": false
  }
}
```

[copy](#)

The `tsconfig.json` file is used to define how the TypeScript compiler should interpret the code, how strictly the compiler should work, which files to watch or ignore, and much more. For now, we will only use the compiler option noImplicitAny, which does not require having types for all variables used.

Let's start by creating a simple Multiplier. It looks exactly as it would in JavaScript.

```
const multiplier = (a, b, printText) => {
  console.log(printText, a * b);
}
```

[copy](#)

```
multiplier(2, 4, 'Multiplied numbers 2 and 4, the result is:');
```

As you can see, this is still ordinary basic JavaScript with no additional TS features. It compiles and runs nicely with `npm run ts-node -- multiplier.ts`, as it would with Node.

But what happens if we end up passing the wrong `types` of arguments to the multiplier function?

Let's try it out!

```
const multiplier = (a, b, printText) => {
  console.log(printText, a * b);
}
```

[copy](#)

```
multiplier('how about a string?', 4, 'Multiplied a string and 4, the result is:');
```

Now when we run the code, the output is: `Multiplied a string and 4, the result is: NaN`.

Wouldn't it be nice if the language itself could prevent us from ending up in situations like this? This is where we see the first benefits of TypeScript. Let's add types to the parameters and see where it takes us.

TypeScript natively supports multiple types including `number`, `string` and `Array`. See the comprehensive list [here](#). More complex custom types can also be created.

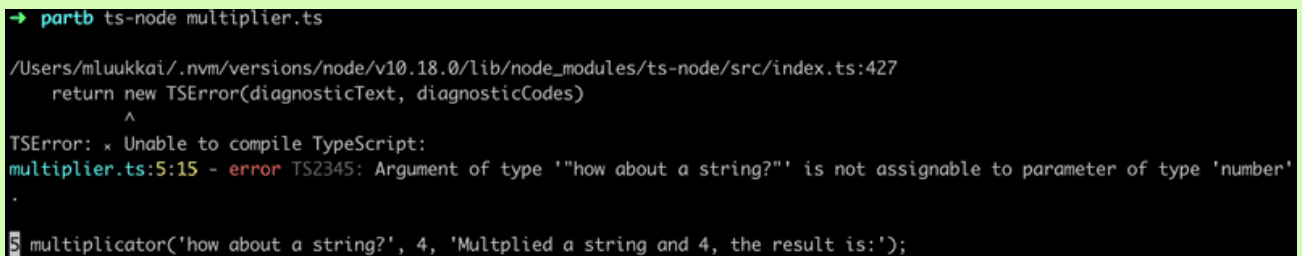
The first two parameters of our function are of type `number` and the last one is of type `string`, both types are [primitives](#):

```
const multiplier = (a: number, b: number, printText: string) => {  
  console.log(printText, a * b);  
}
```

[copy](#)

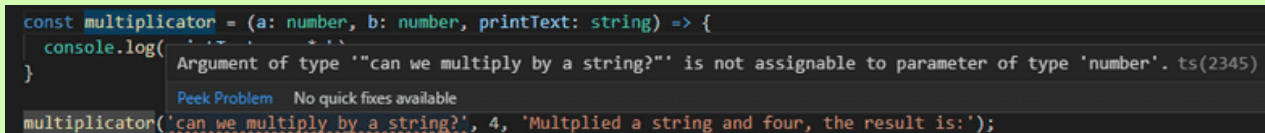
```
multiplier('how about a string?', 4, 'Multiplied a string and 4, the result is:');
```

Now the code is no longer valid JavaScript but in fact TypeScript. When we try to run the code, we notice that it does not compile:



```
→ partb ts-node multiplier.ts  
  
/Users/mluukkai/.nvm/versions/node/v10.18.0/lib/node_modules/ts-node/src/index.ts:427  
    return new TSError(diagnosticText, diagnosticCodes)  
           ^  
TSError: x Unable to compile TypeScript:  
multiplier.ts:5:15 - error TS2345: Argument of type '"how about a string?'" is not assignable to parameter of type 'number'  
.  
5 multiplier('how about a string?', 4, 'Multiplied a string and 4, the result is:');
```

One of the best things about TypeScript's editor support is that you don't necessarily need to even run the code to see the issues. VSCode is so efficient, that it informs you immediately when you are trying to use an incorrect type:



```
const multiplier = (a: number, b: number, printText: string) => {  
  console.log(  
    'Multiplied', a, 'and', b, 'the result is:', a * b  
  );  
}  
  
multiplier('can we multiply by a string?', 4, 'Multiplied a string and four, the result is:');
```

Argument of type '"can we multiply by a string?'" is not assignable to parameter of type 'number'. ts(2345)
Peek Problem No quick fixes available

Creating your first own types

Let's expand our multiplier into a slightly more versatile calculator that also supports addition and division. The calculator should accept three arguments: two numbers and the operation, either `multiply`, `add` or `divide`, which tells it what to do with the numbers.

In JavaScript, the code would require additional validation to make sure the last argument is indeed a string. TypeScript offers a way to define specific types for inputs, which describe exactly what type of input is acceptable. On top of that, TypeScript can also show the info on the accepted values already at the editor level.

We can create a `type` using the TypeScript native keyword `type`. Let's describe our type `Operation`:

```
type Operation = 'multiply' | 'add' | 'divide';
```

copy

Now the `Operation` type accepts only three kinds of values; exactly the three strings we wanted. Using the OR operator `|` we can define a variable to accept multiple values by creating a union type. In this case, we used exact strings (that, in technical terms, are called string literal types) but with unions, you could also make the compiler accept for example both string and number: `string | number`.

The `type` keyword defines a new name for a type: a type alias. Since the defined type is a union of three possible values, it is handy to give it an alias that has a representative name.

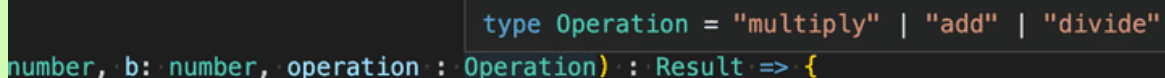
Let's look at our calculator now:

```
type Operation = 'multiply' | 'add' | 'divide';

const calculator = (a: number, b: number, op: Operation) => {
  if (op === 'multiply') {
    return a * b;
  } else if (op === 'add') {
    return a + b;
  } else if (op === 'divide') {
    if (b === 0) return 'can\'t divide by 0!';
    return a / b;
  }
}
```

copy

Now, when we hover on top of the `Operation` type in the calculator function, we can immediately see suggestions on what to do with it:



```
type Operation = "multiply" | "add" | "divide"
number, b: number, operation: Operation): Result => {
```

And if we try to use a value that is not within the `Operation` type, we get the familiar red warning signal and extra info from our editor:

```

s index.ts > ...
1  type Operation = 'multiply' | 'add' | 'divide';
2
3  const calculator = (a: number, b: number, op : Operation) => {
4      if (op === 'multiply') {
5          return a * b;
6      } else if (op === 'add') {
7          return a + b;
8      } else if (op === 'divide') {
9          if (b === 0) return 'can\'t divide by 0!';
10         return a / b;
11     }
12 }
13
14 calculator(1, 2, 'yolo')

```

Argument of type '"yolo"' is not assignable to parameter of type 'Operation'. ts(2345)

[View Problem](#) No quick fixes available

This is already pretty nice, but one thing we haven't touched yet is typing the return value of a function. Usually, you want to know what a function returns, and it would be nice to have a guarantee that it returns what it says it does. Let's add a return value `number` to the calculator function:

```
type Operation = 'multiply' | 'add' | 'divide';
```

copy

```

const calculator = (a: number, b: number, op: Operation): number => {
    if (op === 'multiply') {
        return a * b;
    } else if (op === 'add') {
        return a + b;
    } else if (op === 'divide') {
        if (b === 0) return 'this cannot be done';
        return a / b;
    }
}

```

The compiler complains straight away because, in one case, the function returns a string. There are a couple of ways to fix this:

We could extend the return type to allow string values, like so:

```

const calculator = (a: number, b: number, op: Operation): number | string => {
    // ...
}

```

copy

Or we could create a return type, which includes both possible types, much like our `Operation` type:

```
type Result = string | number;
```

copy

```
const calculator = (a: number, b: number, op: Operation): Result => {
  // ...
}
```

But now the question is if it's `really` okay for the function to return a string?

When your code can end up in a situation where something is divided by 0, something has probably gone terribly wrong and an error should be thrown and handled where the function was called. When you are deciding to return values you weren't originally expecting, the warnings you see from TypeScript prevent you from making rushed decisions and help you to keep your code working as expected.

One more thing to consider is, that even though we have defined types for our parameters, the generated JavaScript used at runtime does not contain the type checks. So if, for example, the `Operation` parameter's value comes from an external interface, there is no definite guarantee that it will be one of the allowed values. Therefore, it's still better to include error handling and be prepared for the unexpected to happen. In this case, when there are multiple possible accepted values and all unexpected ones should result in an error, the switch...case statement suits better than `if...else` in our code.

The code of our calculator should look something like this:

```
type Operation = 'multiply' | 'add' | 'divide';

const calculator = (a: number, b: number, op: Operation) : number => {
  switch(op) {
    case 'multiply':
      return a * b;
    case 'divide':
      if (b === 0) throw new Error('Can\'t divide by 0!');
      return a / b;
    case 'add':
      return a + b;
    default:
      throw new Error('Operation is not multiply, add or divide!');
  }
}

try {
  console.log(calculator(1, 5 , 'divide'));
} catch (error: unknown) {
  let errorMessage = 'Something went wrong: '
  if (error instanceof Error) {
    errorMessage += error.message;
  }
  console.log(errorMessage);
}
```

[copy](#)

Type narrowing

The default type of the catch block parameter `error` is `unknown`. The `unknown` is a kind of top type that was introduced in TypeScript version 3 to be the type-safe counterpart of `any`. Anything is assignable to `unknown`, but `unknown` isn't assignable to anything but itself and `any` without a type assertion or a control flow-based narrowing. Likewise, no operations are permitted on an `unknown` without first asserting or narrowing it to a more specific type.

Both the possible causes of exception (wrong operator or division by zero) will throw an `Error` object with an error message, that our program prints to the user.

If our code would be JavaScript, we could print the error message by just referring to the field `message` of the object `error` as follows:

```
try {
  console.log(calculator(1, 5, 'divide'));
} catch (error) {
  console.log('Something went wrong: ' + error.message);
}
```

[copy](#)

Since the default type of the `error` object in TypeScript is `unknown`, we have to narrow the type to access the field:

```
try {
  console.log(calculator(1, 5, 'divide'));
} catch (error: unknown) {
  let errorMessage = 'Something went wrong: '
  // here we can not use error.message
  if (error instanceof Error) {
    // the type is narrowed and we can refer to error.message
    errorMessage += error.message;
  }
  // here we can not use error.message

  console.log(errorMessage);
}
```

[copy](#)

Here the narrowing was done with the `instanceof` type guard, that is just one of the many ways to narrow a type. We shall see many others later in this part.

Accessing command line arguments

The programs we have written are alright, but it sure would be better if we could use command-line arguments instead of always having to change the code to calculate different things.

Let's try it out, as we would in a regular Node application, by accessing `process.argv`. If you are using a recent npm-version (7.0 or later), there are no problems, but with an older setup something is not right:


```
    throw any
  }
}
```

Cannot find name 'process'. Do you need to install type definitions for node? Try `npm i @types/node`. ts(2580)

Peek Problem No quick fixes available

```
console.log(process.argv)
```

You, a few seconds ago • Uncommitted changes

So what is the problem with older setups?

@types/{npm_package}

Let's return to the basic idea of TypeScript. TypeScript expects all globally-used code to be typed, as it does for your code when your project has a reasonable configuration. The TypeScript library itself contains only typings for the code of the TypeScript package. It is possible to write your own typings for a library, but that is rarely needed - since the TypeScript community has done it for us!

As with npm, the TypeScript world also celebrates open-source code. The community is active and continuously reacting to updates and changes in commonly used npm packages. You can almost always find the typings for npm packages, so you don't have to create types for all of your thousands of dependencies alone.

Usually, types for existing packages can be found from the `@types` organization within npm, and you can add the relevant types to your project by installing an npm package with the name of your package with a `@types/` prefix. For example:

```
npm install --save-dev @types/react @types/express @types/lodash @types/jest
@types/mongoose
```

copy

and so on and so on. The `*@types/**` are maintained by Definitely typed, a community project to maintain types of everything in one place.

Sometimes, an npm package can also include its types within the code and, in that case, installing the corresponding `*@types/**` is not necessary.

NB: Since the typings are only used before compilation, the typings are not needed in the production build and they should always be in the devDependencies of the package.json.

Since the global variable `process` is defined by Node itself, we get its typings from the package `@types/node`.

Since version 10.0 `ts-node` has defined `@types/node` as a peer dependency. If the version of npm is at least 7.0, the peer dependencies of a project are automatically installed by npm. If you have an older npm, the peer dependency must be installed explicitly:

```
npm install --save-dev @types/node
```

copy

When the package `@types/node` is installed, the compiler does not complain about the variable `process`. Note that there is no need to require the types to the code, the installation of the package is enough!

Improving the project

Next, let's add npm scripts to run our two programs `multiplier` and `calculator`:

```
{
  "name": "fs-open",
  "version": "1.0.0",
  "description": "",
  "main": "index.ts",
  "scripts": {
    "ts-node": "ts-node",
    "multiply": "ts-node multiplier.ts",
    "calculate": "ts-node calculator.ts"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "ts-node": "^10.5.0",
    "typescript": "^4.5.5"
  }
}
```

copy

We can get the multiplier to work with command-line parameters with the following changes:

```
const multiplier = (a: number, b: number, printText: string) => {
  console.log(printText, a * b);
}

const a: number = Number(process.argv[2])
const b: number = Number(process.argv[3])
multiplier(a, b, `Multiplied ${a} and ${b}, the result is:`);
```

copy

And we can run it with:

```
npm run multiply 5 2
```

copy

If the program is run with parameters that are not of the right type, e.g.

```
npm run multiply 5 lol
```

[copy](#)

it "works" but gives us the answer:

```
Multiplied 5 and NaN, the result is: NaN
```

[copy](#)

The reason for this is, that `Number('lol')` returns `NaN`, which is actually of type `number`, so TypeScript has no power to rescue us from this kind of situation.

To prevent this kind of behavior, we have to validate the data given to us from the command line.

The improved version of the multiplicator looks like this:

```
interface MultiplyValues {  
  value1: number;  
  value2: number;  
}
```

[copy](#)

```
const parseArguments = (args: string[]): MultiplyValues => {  
  if (args.length < 4) throw new Error('Not enough arguments');  
  if (args.length > 4) throw new Error('Too many arguments');  
  
  if (!isNaN(Number(args[2])) && !isNaN(Number(args[3]))) {  
    return {  
      value1: Number(args[2]),  
      value2: Number(args[3])  
    }  
  } else {  
    throw new Error('Provided values were not numbers!');  
  }  
}
```

```
const multiplicator = (a: number, b: number, printText: string) => {  
  console.log(printText, a * b);  
}
```

```
try {  
  const { value1, value2 } = parseArguments(process.argv);  
  multiplicator(value1, value2, `Multiplied ${value1} and ${value2}, the result is:`);  
} catch (error: unknown) {  
  let errorMessage = 'Something bad happened.'  
  if (error instanceof Error) {  
    errorMessage += ' Error: ' + error.message;  
  }  
}
```

```
console.log(errorMessage);  
}
```

When we now run the program:

```
npm run multiply 1 lol
```

[copy](#)

we get a proper error message:

```
Something bad happened. Error: Provided values were not numbers!
```

[copy](#)

There is quite a lot going on in the code. The most important addition is the function `parseArguments` which ensures that the parameters given to `multiplicator` are of the right type. If not, an exception is thrown with a descriptive error message.

The definition of the function has a couple of interesting things:

```
const parseArguments = (args: string[]): MultiplyValues => {  
  // ...  
}
```

[copy](#)

Firstly, the parameter `args` is an array of strings.

The return value of the function has the type `MultiplyValues`, which is defined as follows:

```
interface MultiplyValues {  
  value1: number;  
  value2: number;  
}
```

[copy](#)

The definition utilizes TypeScript's Interface keyword, which is one way to define the "shape" an object should have. In our case, it is quite obvious that the return value should be an object with the two properties `value1` and `value2`, which should both be of type `number`.

The alternative array syntax

Note that there is also an alternative syntax for arrays in TypeScript. Instead of writing

```
let values: number[];
```

[copy](#)

we could use the "generics syntax" and write

```
let values: Array<number>;
```

[copy](#)

In this course we shall mostly be following the convention enforced by the Eslint rule [array-simple](#) that suggests writing the simple arrays with the `[]` syntax and using the `<>` syntax for the more complex ones, see [here](#) for examples.

Exercises 9.1-9.3

setup

Exercises 9.1-9.7. will all be made in the same node project. Create the project in an empty directory with `npm init` and install the `ts-node` and `typescript` packages. Also, create the file `tsconfig.json` in the directory with the following content:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
  }
}
```

[copy](#)

The compiler option `noImplicitAny` makes it mandatory to have types for all variables used. This option is currently a default, but it lets us define it explicitly.

9.1 Body mass index

Create the code of this exercise in the file `bmiCalculator.ts`.

Write a function `calculateBmi` that calculates a BMI based on a given height (in centimeters) and weight (in kilograms) and then returns a message that suits the results.

Call the function in the same file with hard-coded parameters and print out the result. The code

```
console.log(calculateBmi(180, 74))
```

[copy](#)

should print the following message:

Normal (healthy weight)

copy

Create an npm script for running the program with the command `npm run calculateBmi`.

9.2 Exercise calculator

Create the code of this exercise in file `exerciseCalculator.ts`.

Write a function `calculateExercises` that calculates the average time of `daily exercise hours` and compares it to the `target amount` of daily hours and returns an object that includes the following values:

- the number of days
- the number of training days
- the original target value
- the calculated average time
- boolean value describing if the target was reached
- a rating between the numbers 1-3 that tells how well the hours are met. You can decide on the metric on your own.
- a text value explaining the rating, you can come up with the explanations

The daily exercise hours are given to the function as an array that contains the number of exercise hours for each day in the training period. Eg. a week with 3 hours of training on Monday, none on Tuesday, 2 hours on Wednesday, 4.5 hours on Thursday and so on would be represented by the following array:

```
[3, 0, 2, 4.5, 0, 3, 1]
```

copy

For the Result object, you should create an interface.

If you call the function with parameters `[3, 0, 2, 4.5, 0, 3, 1]` and `2`, it should return:

```
{ periodLength: 7,  
  trainingDays: 5,  
  success: false,  
  rating: 2,  
  ratingDescription: 'not too bad but could be better',  
  target: 2,  
  average: 1.9285714285714286  
}
```

copy

Create an npm script, `npm run calculateExercises`, to call the function with hard-coded values.

9.3 Command line

Change the previous exercises so that you can give the parameters of `bmiCalculator` and `exerciseCalculator` as command-line arguments.

Your program could work eg. as follows:

```
$ npm run calculateBmi 180 91
```

[copy](#)

Overweight

and:

```
$ npm run calculateExercises 2 1 0 2 4.5 0 3 1 0 4
```

[copy](#)

```
{ periodLength: 9,  
  trainingDays: 6,  
  success: false,  
  rating: 2,  
  ratingDescription: 'not too bad but could be better',  
  target: 2,  
  average: 1.7222222222222223  
}
```

In the example, the `first` argument is the target value.

Handle exceptions and errors appropriately. The `exerciseCalculator` should accept inputs of varied lengths. Determine by yourself how you manage to collect all needed input.

A couple of things to notice:

If you define helper functions in other modules, you should use the JavaScript module system, that is, the one we have used with React where importing is done with

```
import { isNotNumber } from './utils';
```

[copy](#)

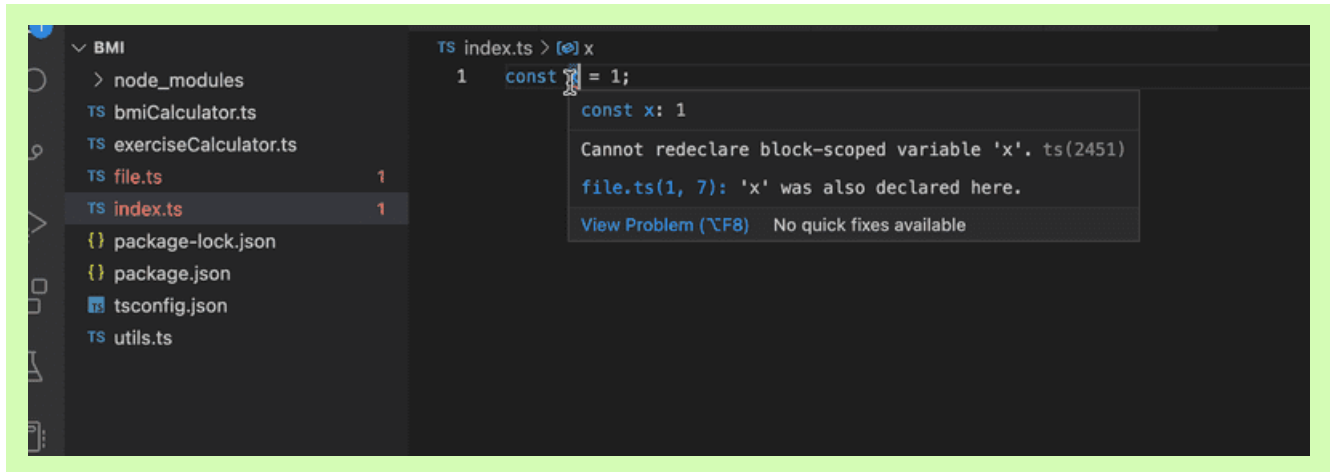
and exporting

```
export const isNotNumber = (argument: any): boolean =>  
  isNaN(Number(argument));
```

[copy](#)

```
export default "this is the default..."
```

Another note: somehow surprisingly TypeScript does not allow to define the same variable in many files at a "block-scope", that is, outside functions (or classes):



This is actually not quite true. This rule applies only to files that are treated as "scripts". A file is a script if it does not contain any export or import statements. If a file has those, then the file is treated as a module, and the variables do not get defined in the block scope.

More about tsconfig

We have so far used only one tsconfig rule noImplicitAny. It's a good place to start, but now it's time to look into the config file a little deeper.

As mentioned, the tsconfig.json file contains all your core configurations on how you want TypeScript to work in your project.

Let's specify the following configurations in our `tsconfig.json` file:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "noImplicitAny": true,
    "esModuleInterop": true,
    "moduleResolution": "node"
  }
}
```

copy

Do not worry too much about the `compilerOptions` ; they will be under closer inspection later on.

You can find explanations for each of the configurations from the TypeScript documentation or from the really handy [tsconfig page](#) , or from the [tsconfig schema definition](#) , which unfortunately is formatted a little worse than the first two options.

Adding Express to the mix

Right now, we are in a pretty good place. Our project is set up and we have two executable calculators in it. However, since we aim to learn FullStack development, it is time to start working with some HTTP requests.

Let us start by installing Express:

```
npm install express
```

[copy](#)

and then add the `start` script to package.json:

```
{
  // ..
  "scripts": {
    "ts-node": "ts-node",
    "multiply": "ts-node multiplier.ts",
    "calculate": "ts-node calculator.ts",
    "start": "ts-node index.ts"
  },
  // ..
}
```

[copy](#)

Now we can create the file `index.ts` , and write the HTTP GET `ping` endpoint to it:

```
const express = require('express');
const app = express();

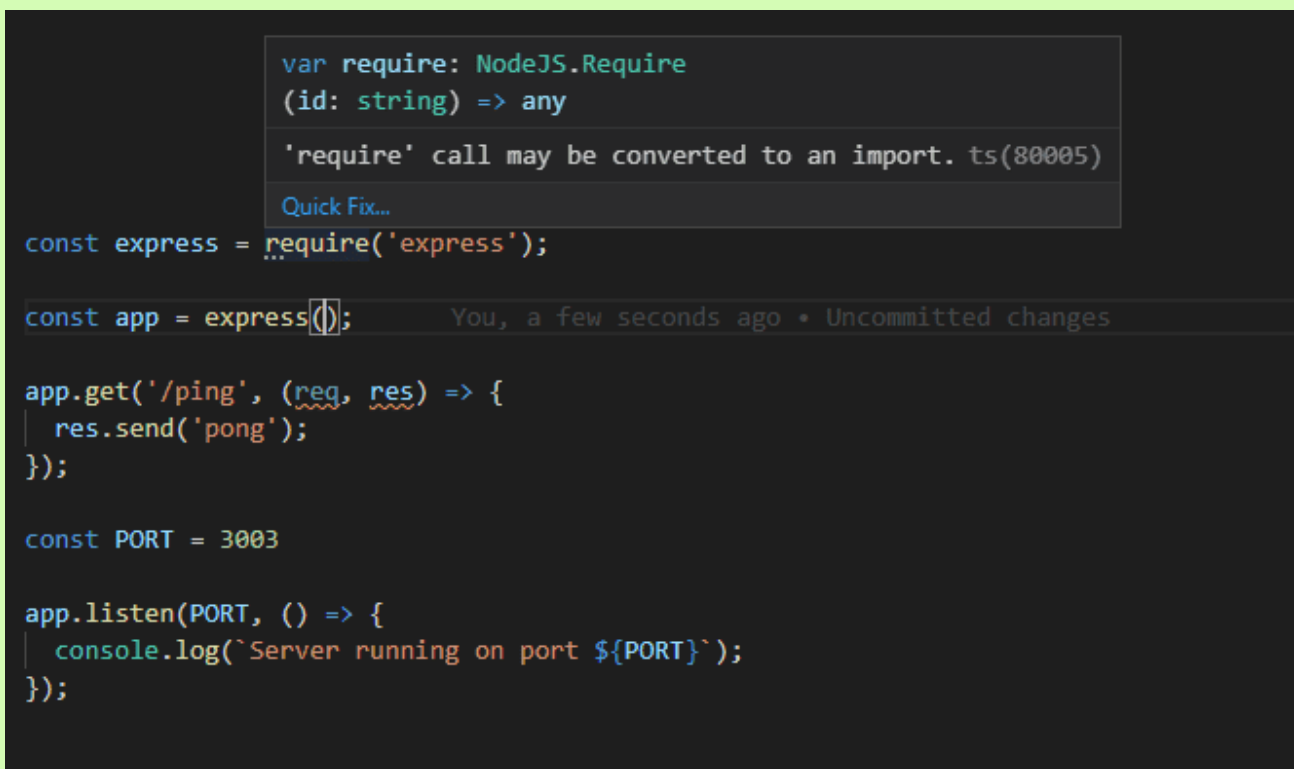
app.get('/ping', (req, res) => {
  res.send('pong');
});

const PORT = 3003;

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

[copy](#)

Everything else seems to be working just fine but, as you'd expect, the `req` and `res` parameters of `app.get` need typing. If you look carefully, VSCode is also complaining about the importing of Express. You can see a short yellow line of dots under `require`. Let's hover over the problem:



The screenshot shows a VS Code editor with a TypeScript file. A hover tooltip is displayed over the `require` function call in the line `const express = require('express');`. The tooltip contains the following text:

```
var require: NodeJS.Require
(id: string) => any
'require' call may be converted to an import. ts(80005)
Quick Fix...
```

The code in the editor is as follows:

```
const express = require('express');

const app = express();

app.get('/ping', (req, res) => {
  res.send('pong');
});

const PORT = 3003

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

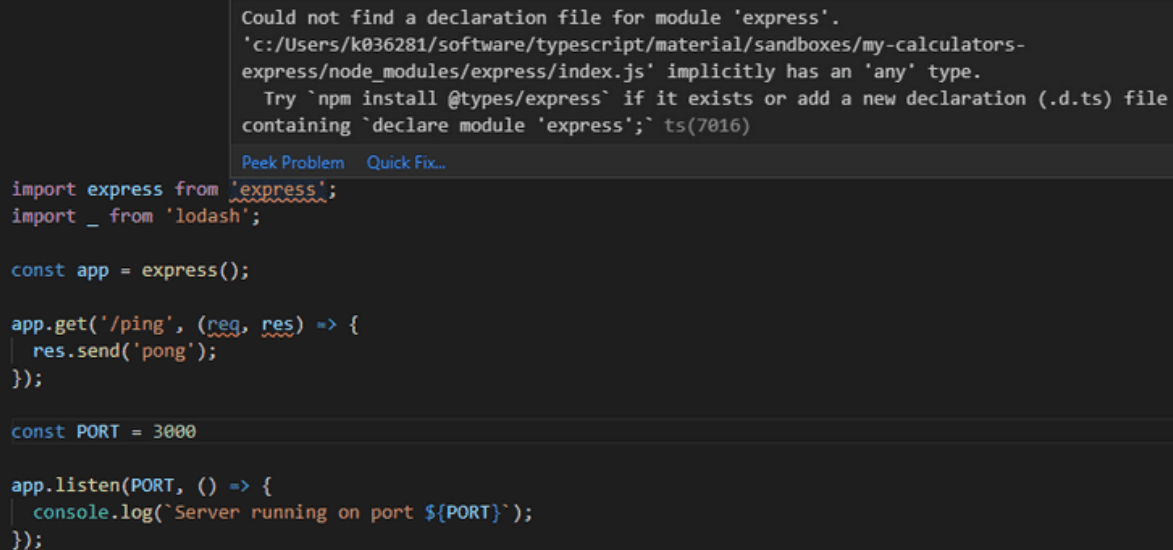
The complaint is that the `'require'` call may be converted to an `import`. Let us follow the advice and write the import as follows:

```
import express from 'express';
```

[copy](#)

NB: VSCode offers you the possibility to fix the issues automatically by clicking the `Quick Fix...` button. Keep your eyes open for these helpers/quick fixes; listening to your editor usually makes your code better and easier to read. The automatic fixes for issues can be a major time saver as well.

Now we run into another problem, the compiler complains about the import statement. Once again, the editor is our best friend when trying to find out what the issue is:



```

Could not find a declaration file for module 'express'.
  'c:/Users/k036281/software/typescript/material/sandboxes/my-calculators-express/node_modules/express/index.js' implicitly has an 'any' type.
  Try `npm install @types/express` if it exists or add a new declaration (.d.ts) file containing `declare module 'express';` ts(7016)
Peek Problem Quick Fix...

import express from 'express';
import _ from 'lodash';

const app = express();

app.get('/ping', (req, res) => {
  res.send('pong');
});

const PORT = 3000

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});

```

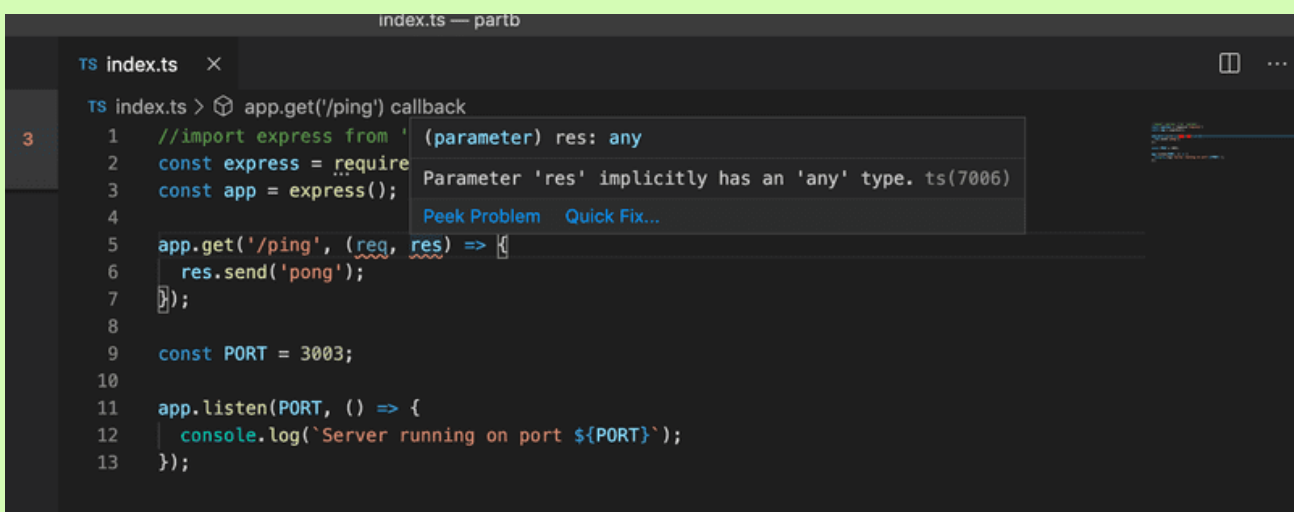
We haven't installed types for `express`. Let's do what the suggestion says and run:

```
npm install --save-dev @types/express
```

copy

And no more errors! Let's take a look at what changed.

When we hover over the `require` statement, we can see that the compiler interprets everything `express`-related to be of type `any`.



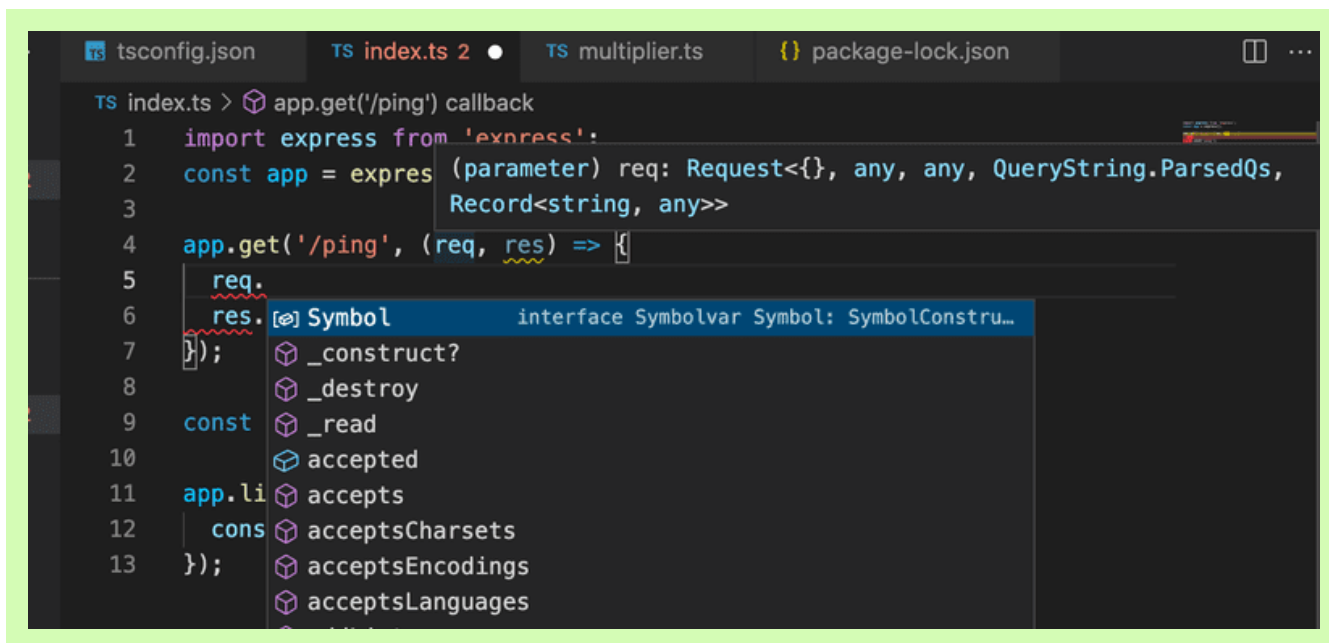
```

index.ts — partb

TS index.ts x
TS index.ts > app.get('/ping') callback
1 //import express from 'express';
2 const express = require('express');
3 const app = express();
4
5 app.get('/ping', (req, res) => {
6   res.send('pong');
7 });
8
9 const PORT = 3003;
10
11 app.listen(PORT, () => {
12   console.log(`Server running on port ${PORT}`);
13 });

```

Whereas when we use `import`, the editor knows the actual types:



Which import statement to use depends on the export method used in the imported package.

A good rule of thumb is to try importing a module using the `import` statement first. We will always use this method in the frontend. If `import` does not work, try a combined method: `import ... = require('...')`.

We strongly suggest you read more about TypeScript modules [here](#).

There is one more problem with the code:



This is because we banned unused parameters in our `tsconfig.json`:

```

{
  "compilerOptions": {
    "target": "ES2022",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "noImplicitAny": true,

```

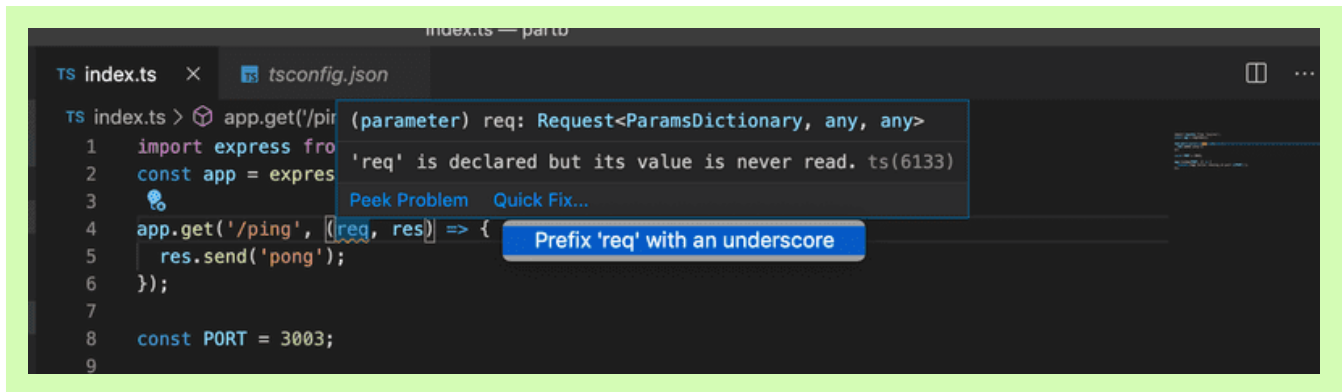
copy

```

    "esModuleInterop": true,
    "moduleResolution": "node"
  }
}

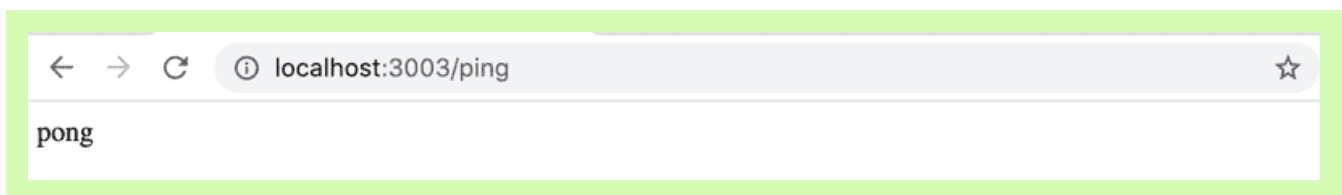
```

This configuration might create problems if you have library-wide predefined functions that require declaring a variable even if it's not used at all, as is the case here. Fortunately, this issue has already been solved on the configuration level. Once again hovering over the issue gives us a solution. This time we can just click the quick fix button:



If it is absolutely impossible to get rid of an unused variable, you can prefix it with an underscore to inform the compiler you have thought about it and there is nothing you can do.

Let's rename the `req` variable to `_req`. Finally, we are ready to start the application. It seems to work fine:



To simplify the development, we should enable `auto-reloading` to improve our workflow. In this course, you have already used `nodemon`, but `ts-node` has an alternative called `ts-node-dev`. It is meant to be used only with a development environment that takes care of recompilation on every change, so restarting the application won't be necessary.

Let's install `ts-node-dev` to our development dependencies:

```
npm install --save-dev ts-node-dev
```

copy

Add a script to `package.json` :

```

{
  // ...
  "scripts": {

```

copy

```
// ...  
"dev": "ts-node-dev index.ts",  
},  
// ...  
}
```

And now, by running `npm run dev`, we have a working, auto-reloading development environment for our project!

Exercises 9.4-9.5

9.4 Express

Add Express to your dependencies and create an HTTP GET endpoint `hello` that answers 'Hello Full Stack!'

The web app should be started with the commands `npm start` in production mode and `npm run dev` in development mode. The latter should also use `ts-node-dev` to run the app.

Replace also your existing `tsconfig.json` file with the following content:

```
{  
  "compilerOptions": {  
    "noImplicitAny": true,  
    "noImplicitReturns": true,  
    "strictNullChecks": true,  
    "strictPropertyInitialization": true,  
    "strictBindCallApply": true,  
    "noUnusedLocals": true,  
    "noUnusedParameters": true,  
    "noImplicitThis": true,  
    "alwaysStrict": true,  
    "esModuleInterop": true,  
    "declaration": true,  
  }  
}
```

[copy](#)

Make sure there aren't any errors!

9.5 WebBMI

Add an endpoint for the BMI calculator that can be used by doing an HTTP GET request to the endpoint `bmi` and specifying the input with query string parameters. For example, to get the BMI of a person with a height of 180 and a weight of 72, the URL is `http://localhost:3003/bmi?height=180&weight=72`.

The response is a JSON of the form:

```
{  
  weight: 72,  
  height: 180,  
  bmi: "Normal (healthy weight)"  
}
```

[copy](#)

See the [Express documentation](#) for info on how to access the query parameters.

If the query parameters of the request are of the wrong type or missing, a response with proper status code and an error message is given:

```
{  
  error: "malformatted parameters"  
}
```

[copy](#)

Do not copy the calculator code to file `index.ts` ; instead, make it a TypeScript module that can be imported into `index.ts` .

The horrors of `any`

Now that we have our first endpoints completed, you might notice that we have used barely any TypeScript in these small examples. When examining the code a bit closer, we can see a few dangers lurking there.

Let's add the HTTP POST endpoint `calculate` to our app:

```
import { calculator } from './calculator';  
  
app.use(express.json());  
  
// ...  
  
app.post('/calculate', (req, res) => {  
  const { value1, value2, op } = req.body;  
  
  const result = calculator(value1, value2, op);  
  res.send({ result });  
});
```

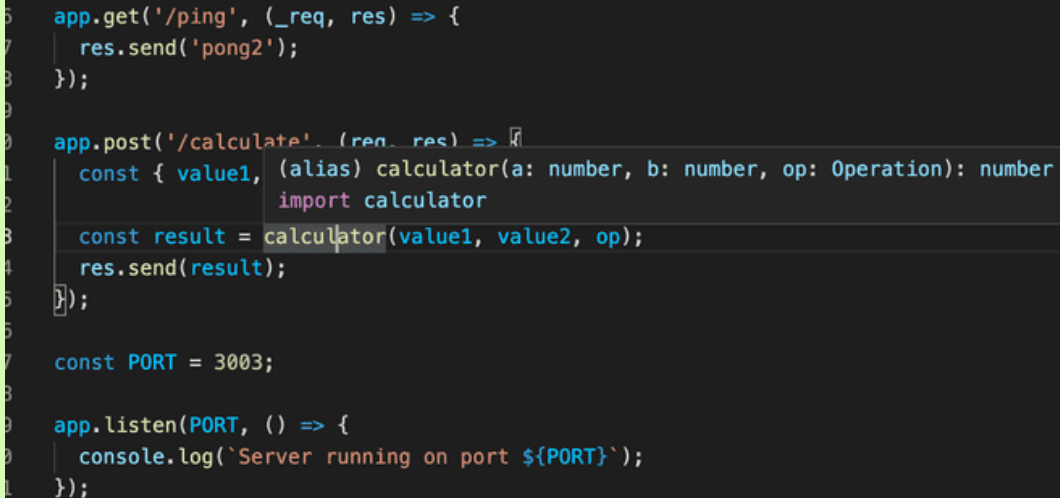
[copy](#)

To get this working, we must add an `export` to the function `calculator` :

```
export const calculator = (a: number, b: number, op: Operation) : number => {
```

copy

When you hover over the `calculate` function, you can see the typing of the `calculator` even though the code itself does not contain any typings:

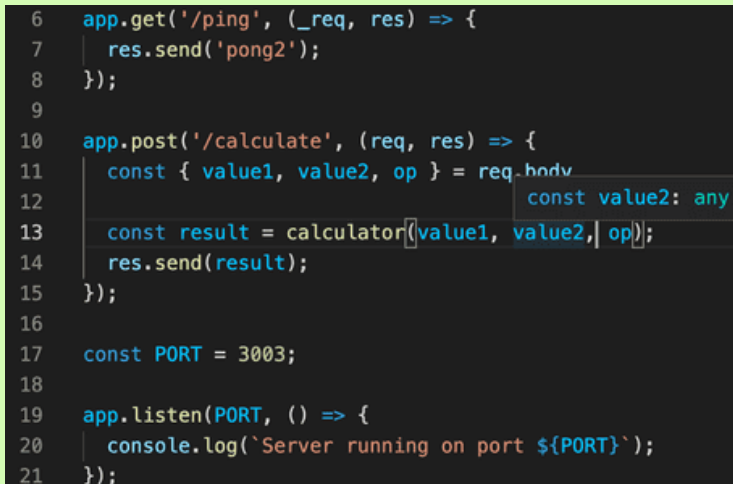


```

6  app.get('/ping', (_req, res) => {
7    res.send('pong2');
8  });
9
10 app.post('/calculate', (req, res) => {
11   const { value1, value2, op } = req.body;
12   const result = calculator(value1, value2, op);
13   res.send(result);
14 });
15
16 const PORT = 3003;
17
18 app.listen(PORT, () => {
19   console.log(`Server running on port ${PORT}`);
20 });

```

But if you hover over the values parsed from the request, an issue arises:



```

6  app.get('/ping', (_req, res) => {
7    res.send('pong2');
8  });
9
10 app.post('/calculate', (req, res) => {
11   const { value1, value2, op } = req.body;
12   const result = calculator(value1, value2, op);
13   res.send(result);
14 });
15
16 const PORT = 3003;
17
18 app.listen(PORT, () => {
19   console.log(`Server running on port ${PORT}`);
20 });
21

```

All of the variables have the type `any`. It is not all that surprising, as no one has given them a type yet. There are a couple of ways to fix this, but first, we have to consider why this is accepted and where the type `any` came from.

In TypeScript, every untyped variable whose type cannot be inferred implicitly becomes of type `any`. `Any` is a kind of "wild card" type, which stands for `whatever` type. Things become implicitly `any` type quite often when one forgets to type functions.

We can also explicitly type things `any`. The only difference between the implicit and explicit `any` type is how the code looks; the compiler does not care about the difference.

Programmers however see the code differently when `any` is explicitly enforced than when it is implicitly inferred. Implicit `any` typings are usually considered problematic since it is quite often due to the coder forgetting to assign types (or being too lazy to do it), and it also means that the full power of TypeScript is not properly exploited.

This is why the configuration rule `noImplicitAny` exists on the compiler level, and it is highly recommended to keep it on at all times. In the rare occasions when you truly cannot know what the type of a variable is, you should explicitly state that in the code:

```
const a : any = /* no clue what the type will be! */.
```

[copy](#)

We already have `noImplicitAny: true` configured in our example, so why does the compiler not complain about the implicit `any` types? The reason is that the `body` field of an Express `Request` object is explicitly typed `any`. The same is true for the `request.query` field that Express uses for the query parameters.

What if we would like to restrict developers from using the `any` type? Fortunately, we have methods other than `tsconfig.json` to enforce a coding style. What we can do is use `ESLint` to manage our code. Let's install ESLint and its TypeScript extensions:

```
npm install --save-dev eslint @typescript-eslint/eslint-plugin @typescript-eslint/parser
```

[copy](#)

We will configure ESLint to disallow explicit any. Write the following rules to `.eslintrc`:

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 11,
    "sourceType": "module"
  },
  "plugins": ["@typescript-eslint"],
  "rules": {
    "@typescript-eslint/no-explicit-any": 2
  }
}
```

[copy](#)

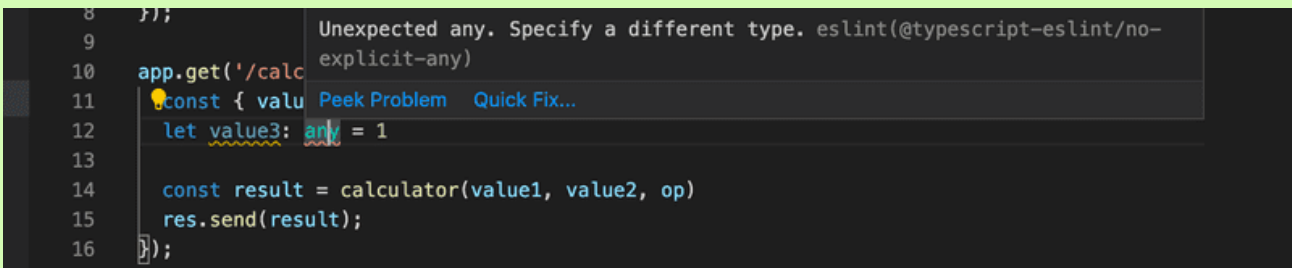
(Newer versions of ESLint have this rule on by default, so you don't necessarily need to add it separately.)

Let us also set up a `lint` npm script to inspect the files with `.ts` extension by modifying the `package.json` file:

```
{
  // ...
  "scripts": {
    "start": "ts-node index.ts",
    "dev": "ts-node-dev index.ts",
    "lint": "eslint --ext .ts ."
  },
  // ...
}
```

copy

Now lint will complain if we try to define a variable of type `any` :



```

8   });
9
10  app.get('/calc', (req, res) => {
11    const { value1, value2, op } = req.query;
12    let value3: any = 1;
13
14    const result = calculator(value1, value2, op);
15    res.send(result);
16  });

```

Unexpected any. Specify a different type. eslint(@typescript-eslint/no-explicit-any)

Peek Problem Quick Fix...

`@typescript-eslint` has a lot of TypeScript-specific ESLint rules, but you can also use all basic ESLint rules in TypeScript projects. For now, we should probably go with the recommended settings, and we will modify the rules as we go along whenever we find something we want to change the behavior of.

On top of the recommended settings, we should try to get familiar with the coding style required in this part and set the semicolon at the end of each line of code to be required.

So we will use the following `.eslintrc`

```
{
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking"
  ],
  "plugins": ["@typescript-eslint"],
  "env": {
    "node": true,
    "es6": true
  },
  "rules": {
    "@typescript-eslint/semi": ["error"],
    "@typescript-eslint/explicit-function-return-type": "off",
    "@typescript-eslint/explicit-module-boundary-types": "off",
    "@typescript-eslint/restrict-template-expressions": "off",
    "@typescript-eslint/restrict-plus-operands": "off",
    "@typescript-eslint/no-unused-vars": [
```

copy

```

    "error",
    { "argsIgnorePattern": "^_" }
  ],
  "no-case-declarations": "off"
},
"parser": "@typescript-eslint/parser",
"parserOptions": {
  "project": "./tsconfig.json"
}
}

```

Quite a few semicolons are missing, but those are easy to add. We also have to solve the ESLint issues concerning the `any` type:



```

1  import express from 'express';
2  const app = express();
3
4  import { calculator } from './calculator';
5
6  app.get('/calculate', (req, res) => {
7    const value2: any = req.query.value2;
8  });
9
10 app.post('/calculate', (req, res) => {
11   const { value1, value2, op } = req.body;
12
13   const result = calculator(Number(value1), Number(value2), op);
14   res.send(result);
15 });
16
17 const PORT = 3003;
18
19 app.listen(PORT, () => {
20   console.log(`Server running on port ${PORT}`);
21 });

```

We could and probably should disable some ESLint rules to get the data from the request body.

Disabling `@typescript-eslint/no-unsafe-assignment` for the destructuring assignment and calling the `Number` constructor to values is nearly enough:

```

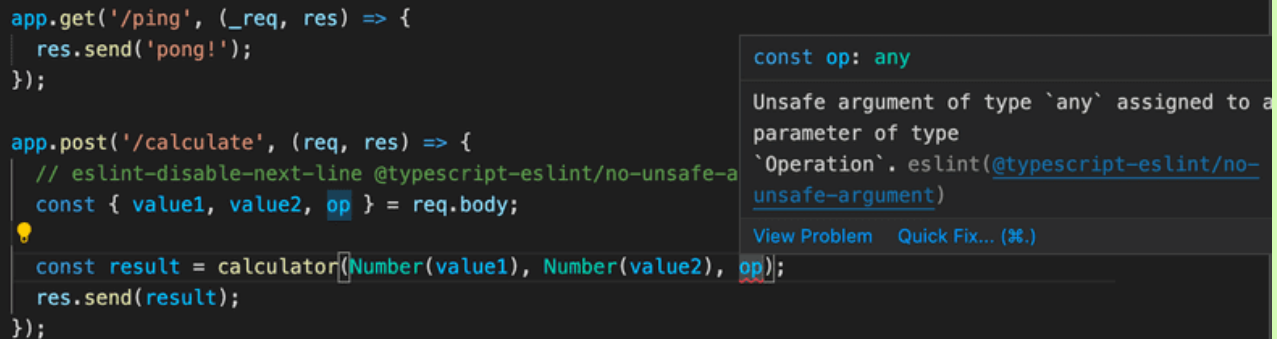
app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const { value1, value2, op } = req.body;

  const result = calculator(Number(value1), Number(value2), op);
  res.send({ result });
});

```

[copy](#)

However this still leaves one problem to deal with, the last parameter in the function call is not safe:



```

app.get('/ping', (_req, res) => {
  res.send('pong!');
});

app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const { value1, value2, op } = req.body;

  const result = calculator(Number(value1), Number(value2), op);
  res.send(result);
});

```

const op: any

Unsafe argument of type `any` assigned to a parameter of type `Operation`. eslint(@typescript-eslint/no-unsafe-argument)

View Problem Quick Fix... (⌘.)

We can just disable another ESLint rule to get rid of that:

```

app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const { value1, value2, op } = req.body;

  // eslint-disable-next-line @typescript-eslint/no-unsafe-argument
  const result = calculator(Number(value1), Number(value2), op);
  res.send({ result });
});

```

copy

We now have ESLint silenced but we are totally at the mercy of the user. We most definitely should do some validation to the post data and give a proper error message if the data is invalid:

```

app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const { value1, value2, op } = req.body;

  if ( !value1 || isNaN(Number(value1)) ) {
    return res.status(400).send({ error: '...' });
  }

  // more validations here...

  // eslint-disable-next-line @typescript-eslint/no-unsafe-argument
  const result = calculator(Number(value1), Number(value2), op);
  return res.send({ result });
});

```

copy

We shall see later in this part some techniques on how the `any` typed data (eg. the input an app receives from the user) can be `narrowed` to a more specific type (such as number). With a proper narrowing of types, there is no more need to silence the ESLint rules.

Type assertion

Using a type assertion is another "dirty trick" that can be done to keep TypeScript compiler and Eslint quiet. Let us export the type `Operation` in `calculator.ts` :

```
export type Operation = 'multiply' | 'add' | 'divide';
```

[copy](#)

Now we can import the type and use a `type assertion` to tell the TypeScript compiler what type a variable has:

```
import { calculator, Operation } from './calculator';

app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const { value1, value2, op } = req.body;

  // validate the data here

  // assert the type
  const operation = op as Operation;

  const result = calculator(Number(value1), Number(value2), operation);

  return res.send({ result });
});
```

[copy](#)

The defined constant `operation` has now the type `Operation` and the compiler is perfectly happy, no quieting of the Eslint rule is needed on the following function call. The new variable is actually not needed, the type assertion can be done when an argument is passed to the function:

```
app.post('/calculate', (req, res) => {
  // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
  const { value1, value2, op } = req.body;

  // validate the data here

  const result = calculator(
    Number(value1), Number(value2), op as Operation
  );

  return res.send({ result });
});
```

[copy](#)

Using a type assertion (or quieting an Eslint rule) is always a bit risky thing. It leaves the TypeScript compiler off the hook, the compiler just trusts that we as developers know what we are doing. If the asserted type does not have the right kind of value, the result will be a runtime error, so one must be pretty careful when validating the data if a type assertion is used.

In the next chapter, we shall have a look at type narrowing which will provide a much more safe way of giving a stricter type for data that is coming from an external source.

Exercises 9.6-9.7

9.6 Eslint

Configure your project to use the above ESlint settings and fix all the warnings.

9.7 WebExercises

Add an endpoint to your app for the exercise calculator. It should be used by doing a HTTP POST request to the endpoint `http://localhost:3002/exercises` with the following input in the request body:

```
{
  "daily_exercises": [1, 0, 2, 0, 3, 0, 2.5],
  "target": 2.5
}
```

[copy](#)

The response is a JSON of the following form:

```
{
  "periodLength": 7,
  "trainingDays": 4,
  "success": false,
  "rating": 1,
  "ratingDescription": "bad",
  "target": 2.5,
  "average": 1.2142857142857142
}
```

[copy](#)

If the body of the request is not in the right form, a response with the proper status code and an error message are given. The error message is either

```
{  
  error: "parameters missing"  
}
```

[copy](#)

or

```
{  
  error: "malformatted parameters"  
}
```

[copy](#)

depending on the error. The latter happens if the input values do not have the right type, i.e. they are not numbers or convertible to numbers.

In this exercise, you might find it beneficial to use the `explicit any` type when handling the data in the request body. Our ESLint configuration is preventing this but you may unset this rule for a particular line by inserting the following comment as the previous line:

```
// eslint-disable-next-line @typescript-eslint/no-explicit-any
```

[copy](#)

You might also get in trouble with rules `no-unsafe-member-access` and `no-unsafe-assignment`. These rules may be ignored in this exercise.

Note that you need to have a correct setup to get the request body; see [part 3](#).

Propose changes to material

[Part 9a](#)
Previous part

Part 9c
[Next part](#)

[About course](#)

[Course contents](#)

[FAQ](#)

[Partners](#)

[Challenge](#)



UNIVERSITY OF HELSINKI

