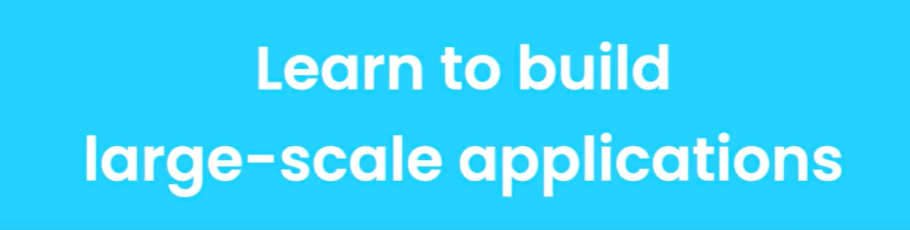


**Welcome**:

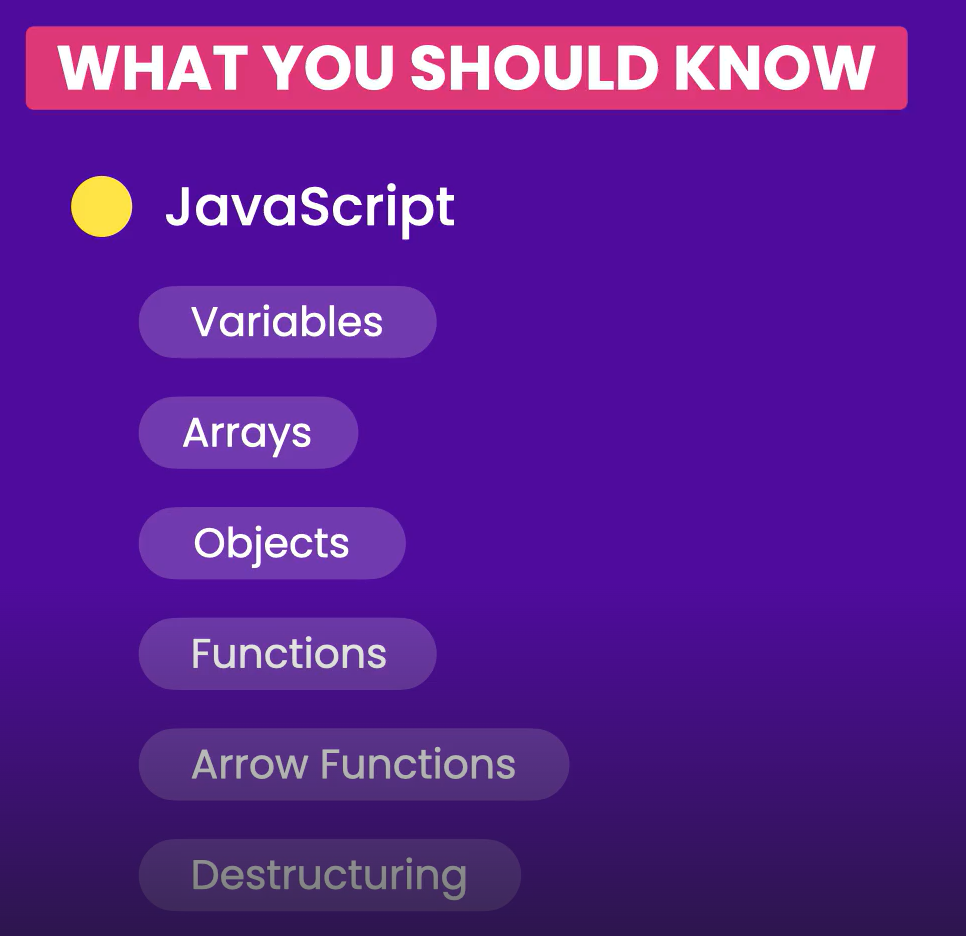
In this course we will learn everything we need to know about TypeScript from basics to more advanced concepts. So by the end of this course we will be able to use TypeScript to…



Everything we need to about TypeScript is in one place, so we don’ have to jump between different tutorials back and forth.

**Prerequisites**:

Since TypeScript is built on top of JavaScript…



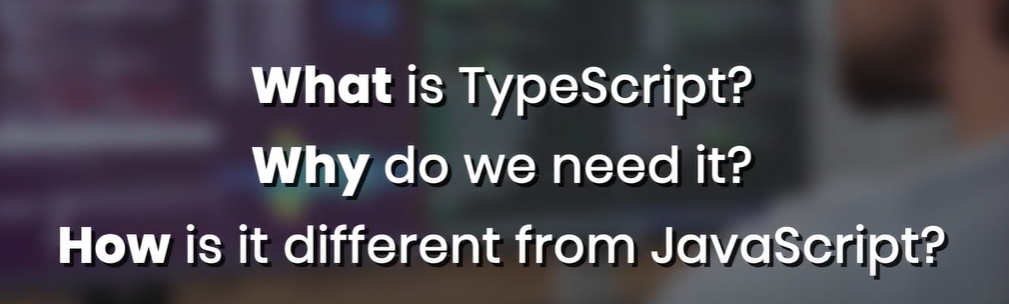
**Introduction**:

In this section we will learn about…



**What is TypeScript**:

In this lesson we will learn…



So to answer the first question.

TypeScript is…

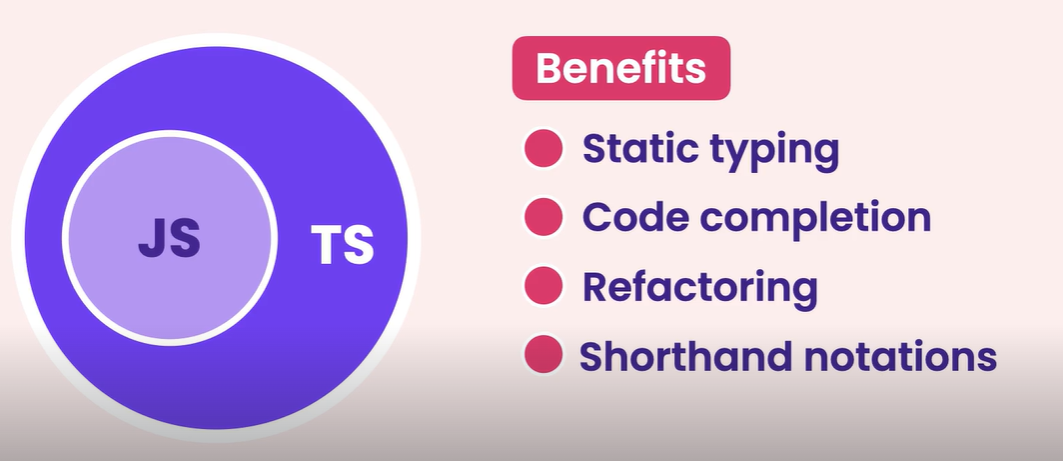


And it was created at Microsoft.

We can think of this as a brother or sister or JavaScript. **“***JavaScript is like a kid without any discipline, who does whatever he/she wants. TypeScript on the other hand is like a kid with some discipline***”**.

*Technically speaking TypeScript is a programming language, built on top of JavaScript, so every JavaScript file is a valid TypeScript file*.

And it adds some really cool features in JavaScript,

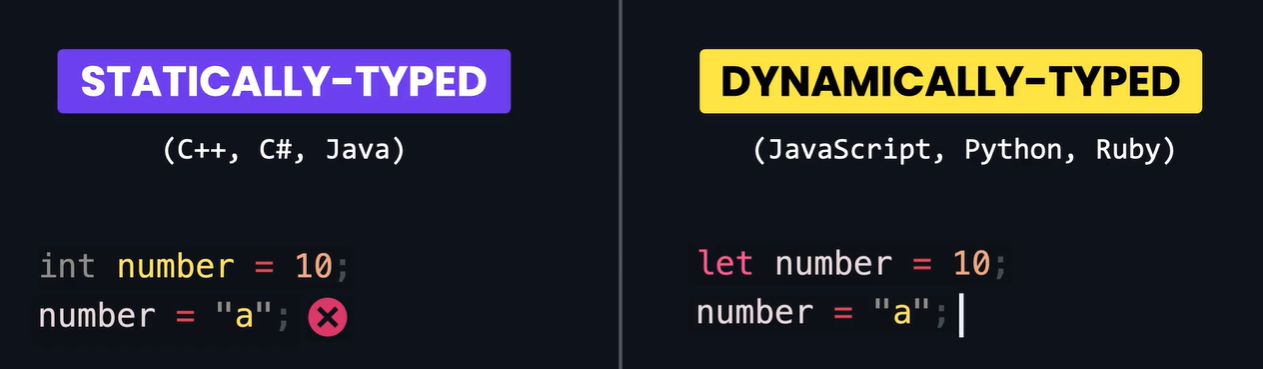


That helps us build more robust and maintainable applications in less time.

The most important feature TS offers is *static typing*.

What does it mean?

We have two different types of languages available.



In Statically typed language like Java, we know the type of variables at compile time or while coding. For example variable,

int number = 10;

Can only hold integer values, nothing else. So we cannot set it to a string or another type of object.

In Dynamically typed language like JavaScript, the type of variable is dynamic. So its determined at run time and it can also change. For example,

let number = 10;

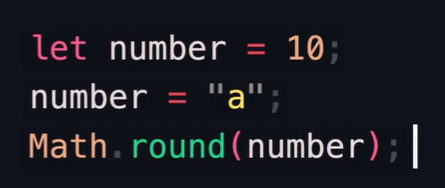
is a number initially but we can also change it string

number = ‘decimal’;

So we can say this variable does not have a fixed or static type, the type is determined and can be changed at run time.

*This feature gives a lot of flexibility but can also lead to some problems*.

What if pass this variable to a function, that expects a number.



Then our application might misbehave or crash.

The problem is that we won’t know about these issues until we run our application or our unit tests (*assuming we have them in first place*).

So *we have to test every function with various edge cases to catch these bugs and this is the problem that typescript tries to solve*.

We can also say…



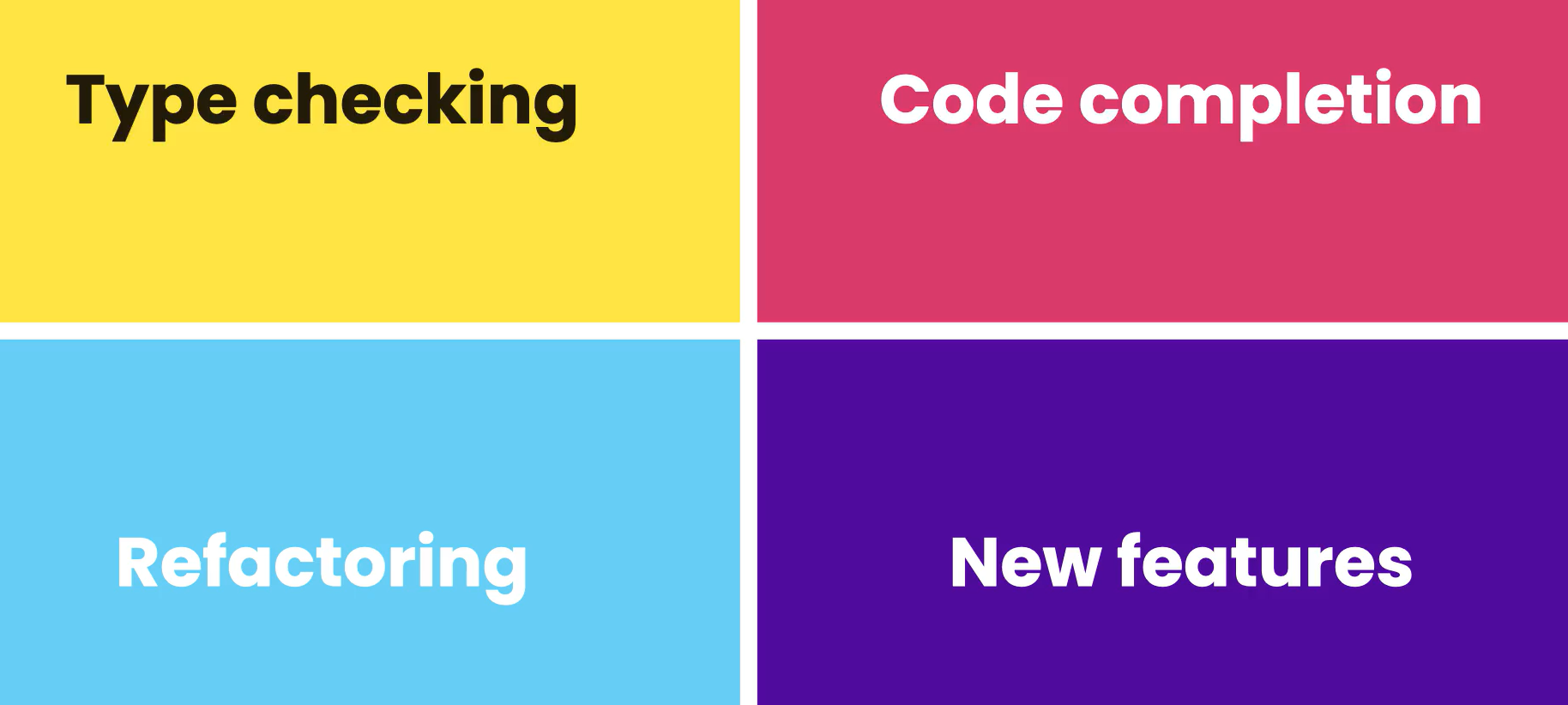
With TS we explicitly set the type of our variables upon declaration, just how we code in statically typed languages.

Then *we pass our code to TS compiler and compiler will tell us if we are doing something wrong, so we can catch our mistake at compile time rather than run time*.



So if we set a variable to a number, we cannot set it to a string. TypeScript compiler is going to stop us right there and we don’t even have to run our application or our unit tests to find them.

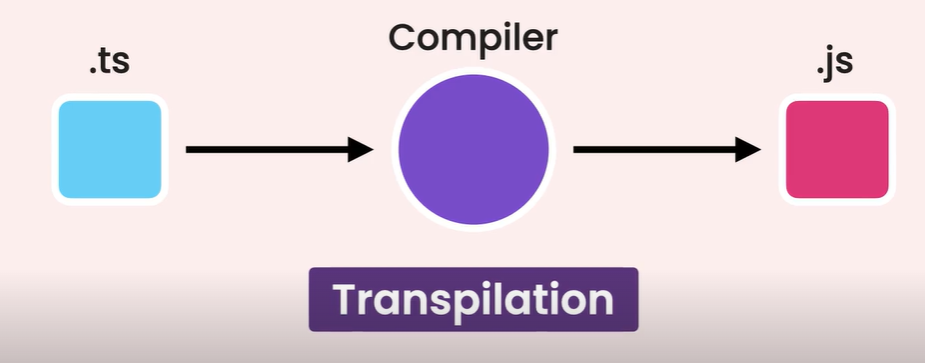
But TS is more then type checking, most code editors these days have great support for typescript, so they can detect the type of our variables and offer productivity boosting features like *code completion* and *refactoring*.



Also TS includes additional features that help us write cleaner and more concise code.

All these great benefits aside TS have some drawbacks as well,

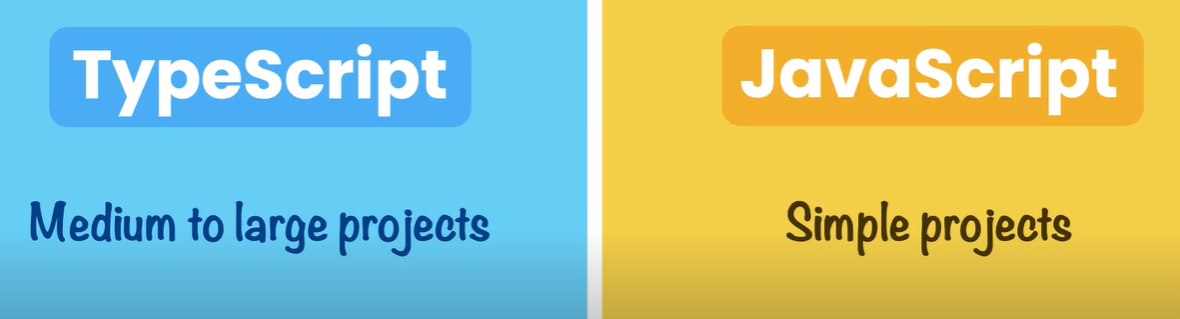
🡪 First with TS there is always a compilation step involved, because at this time browsers don’t understand TypeScript code.



So we have to give our code to TS compiler to compile and translate into JavaScript. This process is called *Transpilation*.

🡪 Second, we have to be a bit more disciplined in writing code.

So here is where you use TS and JS…



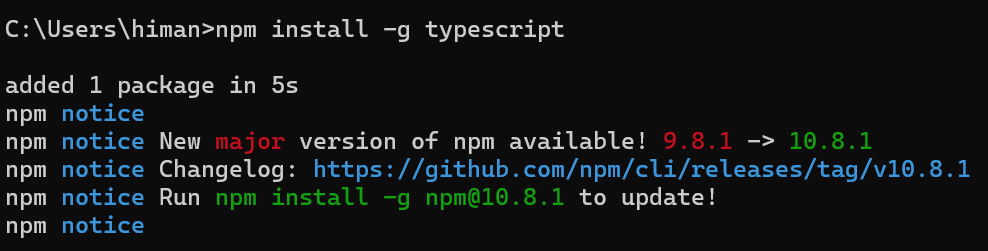
**Setting up Development Environment**:

1. Node.

Since we are going to use node package manager or npm to install the TS compiler.

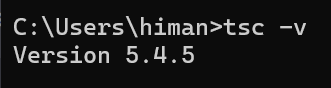
Then use,

npm install -g typescript (*to install it globally*)



1. To check the TS version,

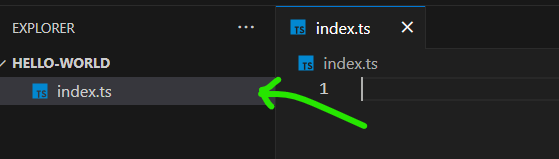
tsc -v



1. VS code.

**First TypeScript program**:

Create a new folder and inside it create a new file called *index.ts*.



Earlier we said typescript is built on top of JavaScript, so we can also say that *TS is a superset of JS*, which means it has everything JS has and some additional features.

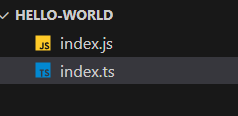
So we can write any JavaScript code inside it and it will be a valid TypeScript code.

console.log('Hello World')

And in the terminal, type:

tsc .\index.ts

Now let’s open our project folder and we see index.js which is a result of compilation.



We see the exact same code in index.js because in index.ts we have not used any typescript features.

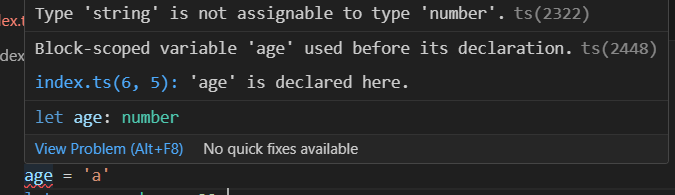
So let’s write a bit of TS code, in index.ts.

let age: number = 20;

Here we declared a variable called *age* and annotated it with a number (*or* *explained a variable using : colon*) and then initialized it to 20.

Here is the beautiful part, since we have declared age as a number, we cannot set it to a string or another type of object.

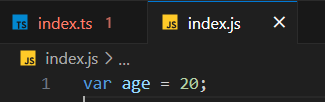
Look we got, this error right here…



Now let’s remove this bad line and recompile our file, using

tsc .\index.ts

And take a look over in index.js



This is the JavaScript code that typescript compiler generated.

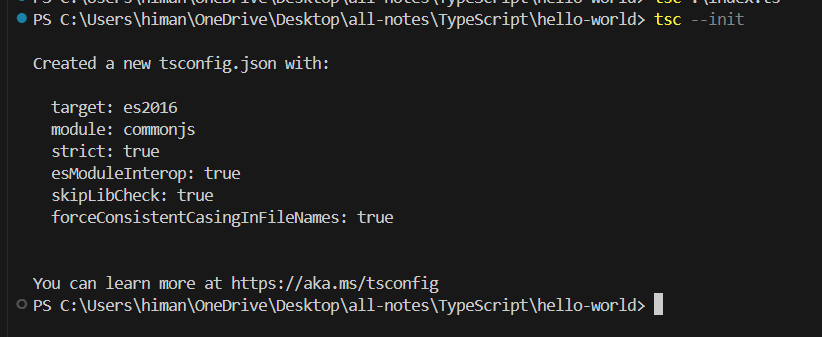
So instead of let keyword here, we have var. Because by default the TS compiler uses an older version of JS called ES5.

In next lesson we will learn how to configure TS compiler to target a newer JS version. So the code that will be generated will be more modern.

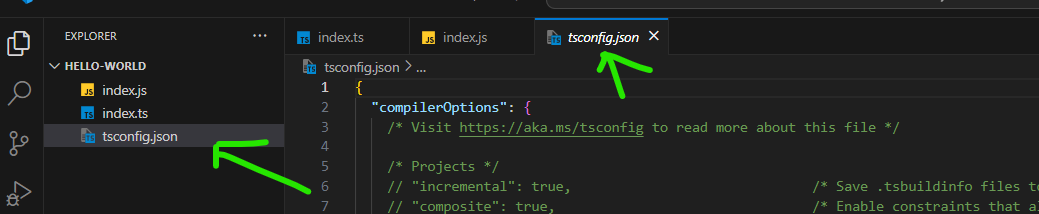
**Configuring TS compiler**:

Let’s create a configuration file for the typescript compiler. In the terminal,

tsc --init



It will create a new configuration file called *tsconfig.json* with these settings.



In these files we have a number settings and most of them are commented out by default but we are only going to use a couple of them.

If curious about any of them, we have the description in front of these settings.

    // "rootDir": "./",                                  /\* Specify the root folder within your source files. \*/

    // "moduleResolution": "node10",                     /\* Specify how TypeScript looks up a file from a given module specifier. \*/

    // "baseUrl": "./",                                  /\* Specify the base directory to resolve non-relative module names. \*/

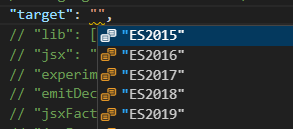
    // "paths": {},                                      /\* Specify a set of entries that re-map imports to additional lookup locations. \*/

First one is *target*,

"target": "es2016",

It specifies the version of JavaScript which TS compiler is going to generate. Depending on where we want to deploy our application, we can use a higher target(*resulting in shorter and concise code*).

By pressing **ctrl + space** we can see all valid values.



Next setting we see is *module*(*will talk about it later when we talk about modules*).

    "module": "commonjs",

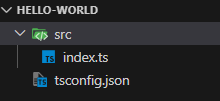
In the module section, we have a setting called *rootDir*, which specifies the directory containing the source files.

    "module": "commonjs",                                /\* Specify what module code is generated. \*/

    "rootDir": "./", *🡪 by default set to current*                                   
/\* Specify the root folder within your source files. \*/

We can create a new folder in our project folder and name it *src*.

"rootDir": "./src",



We have a similar setting called *outDir*, This specifies the directory which will contain our JavaScript files. Let’s change to *./dist*.

This means when we compile our code using the TS compiler, our JS files will be stored in *dist* or distributable folder.

 "outDir": "./dist",

Another useful setting is *removeComments*, If we enable this, The TS compiler is going to remove all the comments that we add in our code, so the generated JavaScript code is going to be shorter.

    "removeComments": true,                           /\* Disable emitting comments. \*/

Another useful setting in this section is *noEmitOnError*, So By default when we compile our code even if we have errors in our code, the TS compiler will still generate JS files.

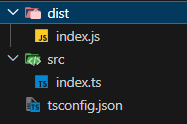
    "noEmitOnError": true,                            /\* Disable emitting files if any type checking errors are reported. \*/

So a good idea is to enable this setting to disable generating JS files in case we have any error in TS files.

With this tsconfig.json file in place, we can now compile all TS files in this project just using *tsc* command.



And we have a new dist folder which contains our JS files.



**Debugging the TS applications**:

This is very useful when things go wrong and our code does not work as expected. So we can run our code line by line and see what exactly happens under the hood.

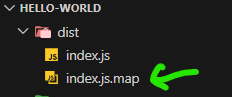
To do this we need to follow few steps:

1. In tsconfig.json, in the *emit* section we enable the *sourceMap* feature.

    "sourceMap": true,                                /\* Create source map files for emitted JavaScript files. \*/

A source map is a file that specifies how each line of our TS code maps to the generated JS code.

To see it in action, recompile our code using tsc. We see a new file in our dist folder called *index.js.map* (*our source map*).



Inside this file,

{"version":3,"file":"index.js","sourceRoot":"","sources":["../src/index.ts"],"names":[],"mappings":";AAAA,IAAI,GAAG,GAAW,EAAE,CAAC"}

This is for debuggers to understand not us.

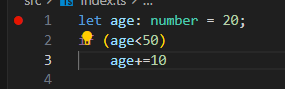
To make debugging more interesting, let’s add some logic in index.ts file.

let age: number = 20;

if (age<50)

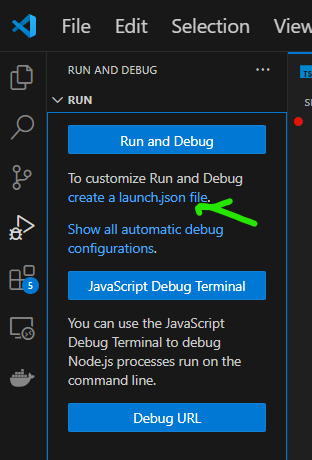
    age+=10

Now we will insert a breakpoint here,

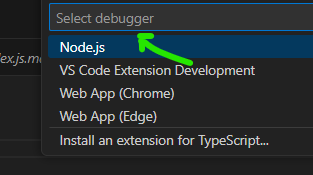


So when we start debugging execution stops right at this line / breakpoint. From this point forward, we can execute our code line by line.

Now we go to the debug panel,

 and click on *create a launch.json file*.

It will open a dropdown from where we will select Node.js



It creates this file called launch.json in our project.

{

    // Use IntelliSense to learn about possible attributes.

    // Hover to view descriptions of existing attributes.

    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387

    "version": "0.2.0",

    "configurations": [

        {

            "type": "node",

            "request": "launch",

            "name": "Launch Program",

            "skipFiles": [

                "<node\_internals>/\*\*"

            ],

            "program": "${workspaceFolder}\\src\\index.ts",

            "outFiles": [

                "${workspaceFolder}/\*\*/\*.js"

            ]

        }

    ]

}

It is a JSON file with some configuration that tells VS code how to debug this application.

Here,

"program": "${workspaceFolder}\\src\\index.ts",

*//our source files are stored here*

And our output files are stored in our workspace in files with JS extension,

            "outFiles": [

                "${workspaceFolder}/\*\*/\*.js"

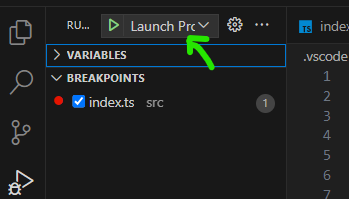
            ]

Let’s add a new setting called *preLaunchTask*:

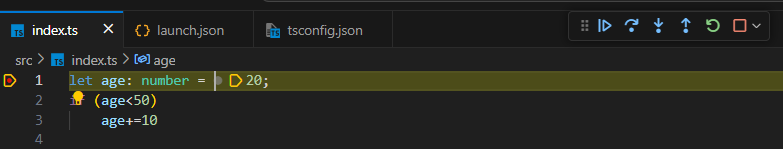
"preLaunchTask": "tsc: build - tsconfig.json",

With this setting, we are telling VS code to use the TS compiler to build our application using this configuration file.

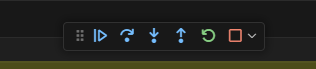
Now to start debugging, go to debug panel and click on ‘*Launch Program*’ (*the label that we saw in launch.json*)



The execution is stopped right on the breakpoint,



On the top toolbar, we have couple of tools for executing our code.



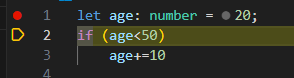
 🡪 *Step over*: For executing one line. (shortcut: **F10**)

🡪 *Step into*: For stepping into a function.(shortcut: **F11**)

🡪 *Step out*: For stepping out of the function.(shortcut: shift + F11)

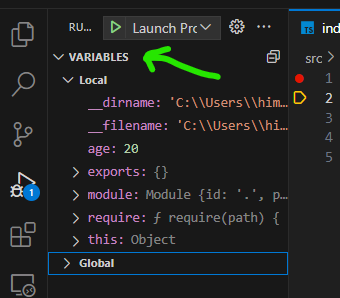


As soon as we step over the breakpoint,



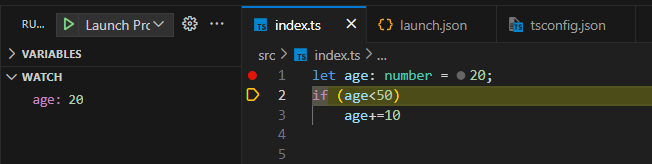
Our line gets executed.

On the left hand side,

🡨In variables we see all the variables that are detected in this debugging session.

We can see age variable that is set to 20 and as we execute each line, we see the value of this variable getting updated.

Note: If we want to see a specific variable only, we can go to watch window and insert a watch.



To see the updated value of age, we need to add a console.log(age) statement at the end and keep stepping over until getting there.

