TypeScript is a **JavaScript Superset** aka it is a language building up on JavaScript.
It adds **new features and advantages** to JS.

TypeSrcipt can't be executed by javascript environments like the browser. Node.js can't either.

TypeScript is compiled to JavaScript which allows it to be run in these environments.

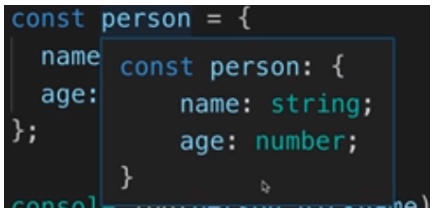
The TypeScript compiler compiles these **"new features"** to **JavaScript work-arounds.**
It is technically writeable in JS, but TypeScript makes it easier.

Nicer syntax for complex JS snippet.

TypeScript adds types.
Helps catch errors earlier in development.
Extra error checking

TYPES

- **number** - *1, 5.3, -10* - all numbers, no differentiation between integers and floats
- **string** - *'hello', "world" `hello`* - all text values. simple quotation marks and object literals/ interpolation
- **boolean** - *true/false* - not truthy values like 0 or null
- **object** - typical object

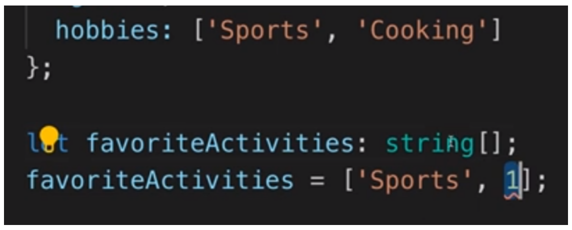


```
const person = {
  name
  age:
};
```

```
const person: {
    name: string;
    age: number;
}
```

this is a concrete object, not a JS object. rather than a key-value pairing, it has a key-type pairing.

- **array** - [1, 2, 3] - any JS array. type can be flexible or strict



```
  hobbies: ['Sports', 'Cooking']
};

let favoriteActivities: string[];
favoriteActivities = ['Sports', 1];
```

hobbies is being defined as an array containing those 2 string values.

this is how we would set a variable to only hold an array of strings as its value.
this means that this mix of string and number would cause an error as it is being shown in the image.

if we wanted to create an **array that could take any data type**, we would use the keyword any in place of the word string. this would look like **"any[ ]"**.

we rarely use the "any" keyword, because it defeats the strictness of TS.

- **tuple** - *[1, 2]* - an array with a fixed length
- **enum** - *enum{ NEW, OLD }* -

TYPE ALIASES

Type aliases can be used to "create" your own types. You're not limited to storing union types though - you can also provide an alias to a (possibly complex) object type.

For example:

```
type User = { name: string; age: number };
const u1: User = { name: 'Max', age: 30 }; // this works!
```

This allows you to avoid unnecessary repetition and manage types centrally.

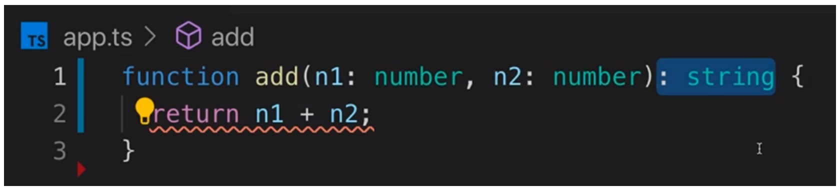
For example, you can simplify this code:

```
function greet(user: { name: string; age: number }) {
  console.log('Hi, I am ' + user.name);
}

function isOlder(user: { name: string; age: number },
checkAge: number) {
  return checkAge > user.age;
}
```

To:

```
type User = { name: string; age: number };

function greet(user: User) {
  console.log('Hi, I am ' + user.name);
}

function isOlder(user: User, checkAge: number) {
  return checkAge > user.age;
}
```

RETURN TYPES

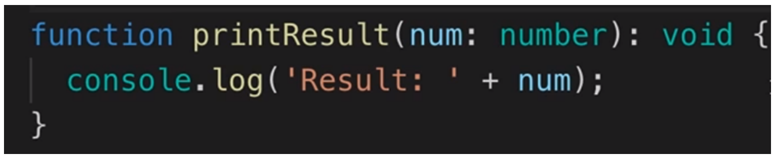

app.ts > add

```
function add(n1: number, n2: number): string {
  return n1 + n2;
}
```

The highlighted part is a return type. return types just let you explicitly say what type of output/return you want from a function. Here it is causing an error because typescript knows that we are taking in two numbers and adding them together so the output would be a number and not a string.

Apart from the previously listed types in TS, we also have **void** and **undefined.**

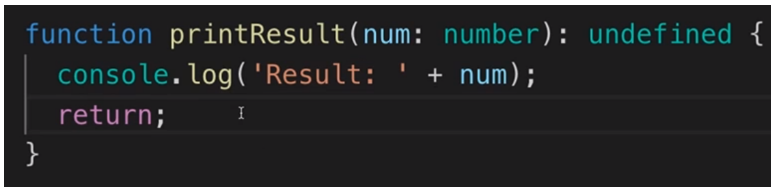
void - *void* - adding this as the return type of a function tells TS that the function will not have a return statement at all. It does not return anything.



```
function printResult(num: number): void {
  console.log('Result: ' + num);
}
```

Note: No return statement

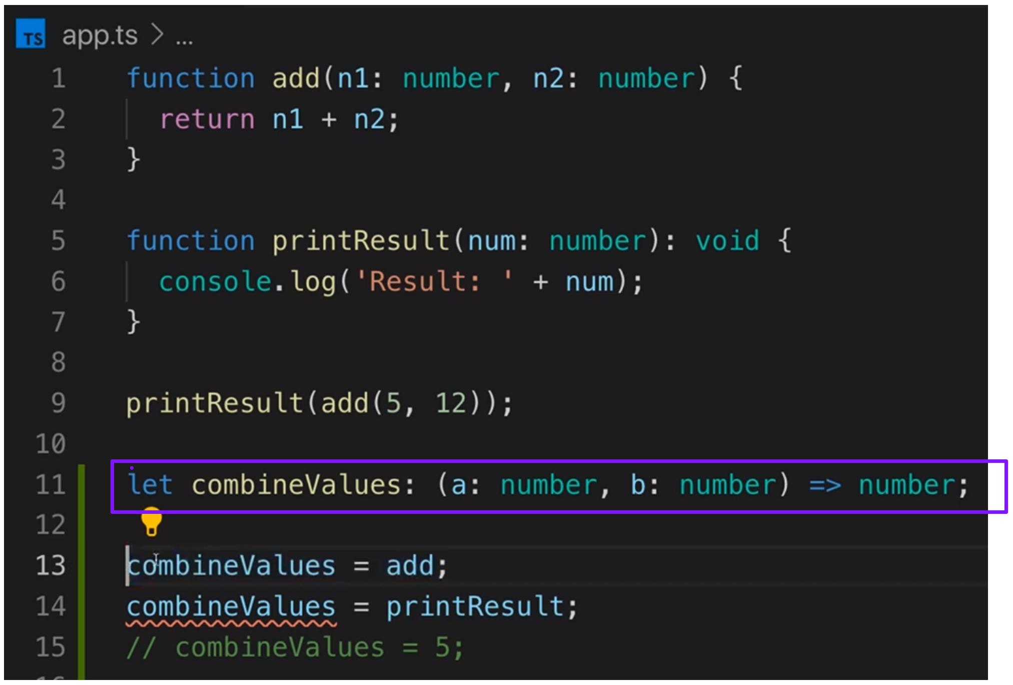
undefined - *undefined* - adding this as the return type of a function tells TS that the function will have an empty return statement/ a return statement that is equal to undefined.



```
function printResult(num: number): undefined {
  console.log('Result: ' + num);
  return;
}
```

Note: Return statement, but empty

FUNCTIONS AS TYPES



app.ts > ...

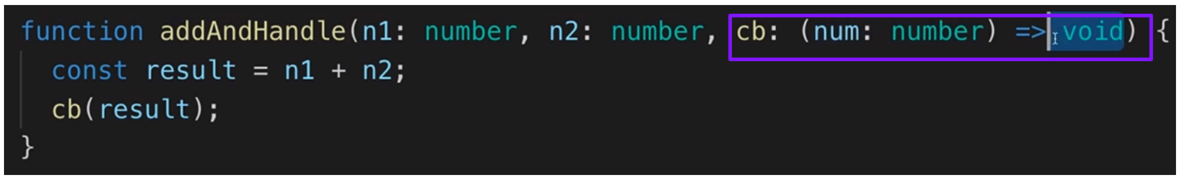
```
function add(n1: number, n2: number) {
  return n1 + n2;
}

function printResult(num: number): void {
  console.log('Result: ' + num);
}

printResult(add(5, 12));

let combineValues: (a: number, b: number) => number;

combineValues = add;
combineValues = printResult;
// combineValues = 5;
```

we are defining combinedValues as a function that takes 2 numbers and returns a number.

This allows us to assign the add function as the value of combinedValues because it follows the rules given by combinedValues. It takes in 2 parameters, which are both numbers, and its implicit return value is a number.

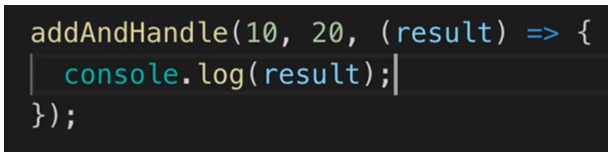
This also now gives us a warning when compiling because we are trying to assign combineValues to a function that doesn't follow the rules stated by the variable.

FUNCTION AS TYPE USED FOR CALLBACK ARG



```
function addAndHandle(n1: number, n2: number, cb: (num: number) => void) {
  const result = n1 + n2;
  cb(result);
}
```

It is exactly as shown above, just passed as a callback function into a function. This means that when we call the addAndHandle function later, we need to pass it 2 numbers and a callback function that takes a number and has no explicit return or returns undefined.

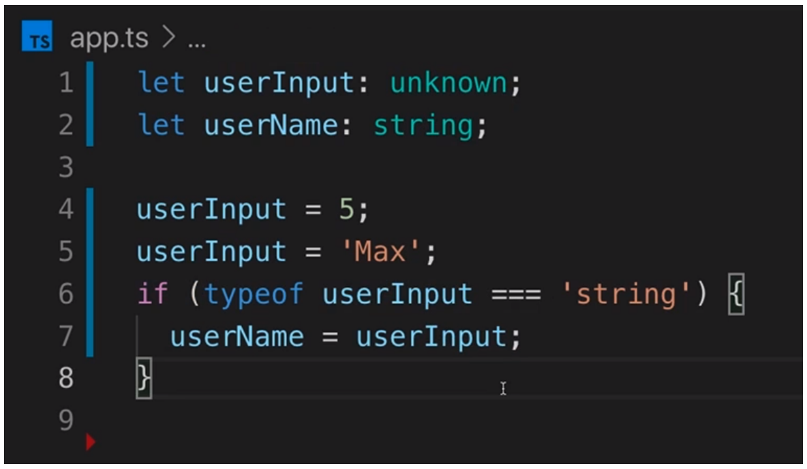


```
addAndHandle(10, 20, (result) => {
  console.log(result);
});
```

Here we are passing the function the 2 number args and then a callback which follows all of the rules that we defined in the original function. It takes a single number arg, and does not have an explicit return.

2 MORE TYPES

unknown - *unknown* - is kind of like **any**, but less flexible (which is good).



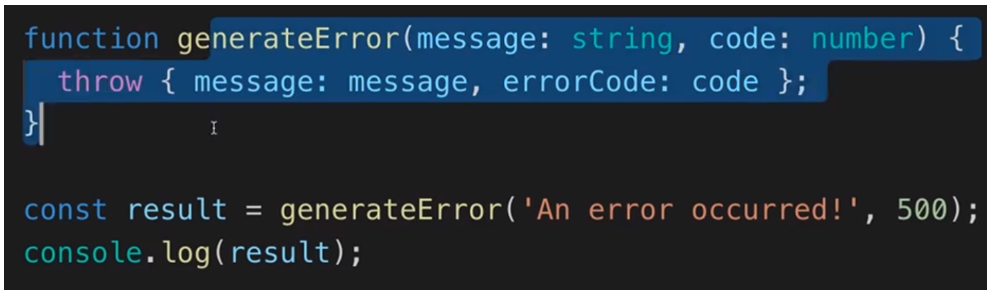
app.ts > ...

```
let userInput: unknown;
let userName: string;

userInput = 5;
userInput = 'Max';
if (typeof userInput === 'string') {
  userName = userInput;
}
```

1. we program userInput to accept any value, by assigning unknown to it. but because of this, we don't know which value type it will be carrying at any given point, because we didn't restrict it to a string or a number or even a union type.
2. because we don't know what value type userInput is carrying, or it is unknown, we can't assign the value of userInput as the value of another variable, userName, which does have restrictions to hold only string values. TS cannot assure that the value of userInput will be a string because with unknown, it can be any type.
3. **any** just disables all type checking and makes TS say "I give up, do whatever you want." This is

3. why even if the value of userInput is set to a number value, and userName is expecting a string, userName will still allow that number type variable to be assigned to it

never - *never* - it tells us that the function will never even make it to the return stage.



```
function generateError(message: string, code: number) {
  throw { message: message, errorCode: code };
}

const result = generateError('An error occurred!', 500);
console.log(result);
```

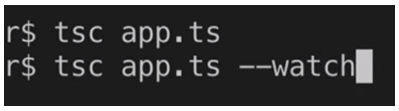
This generateError function is a perfect example of a function that has a return type of never. Because the function simply takes two args and uses them to generate/throw a new error. The error interrupts the ending of the function. So it never even gets to the return stage.

The console.log(result) will show nothing in the console. Because again, not even undefined is returned. It never reaches that stage.

TS COMPILER

WATCH MODE

Instead of the typical **tsc <filename>** to compile our TS code into JS, we want to add --watch at the end of the command:



```
r$ tsc app.ts
r$ tsc app.ts --watch
```

NOTE: it can also be written as: **tsc <filename> -w**

COMPILING AN ENTIRE PROJECT RATHER THAN A SINGLE FILE

```
$ tsc --init
```

Instead of writing **tsc <filename>**, you initialize a tsc compiler in your project folder:

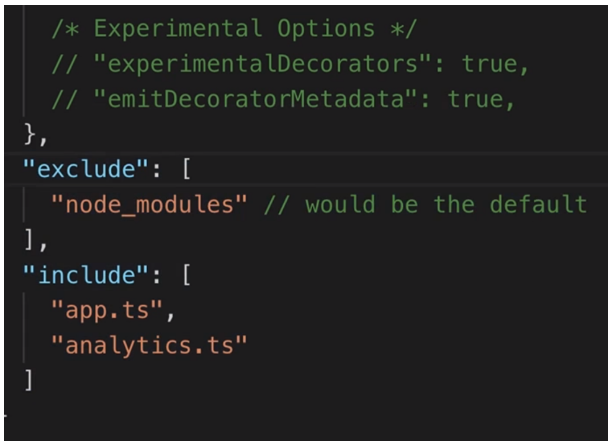
tsc --init

After this, you can just run **tsc** or **tsc -w** if you want watch mode turned on too.

TSCONFIG.JSON

The ts.config file tells TS how it should compile our files.

INCLUDE AND EXCLUDE

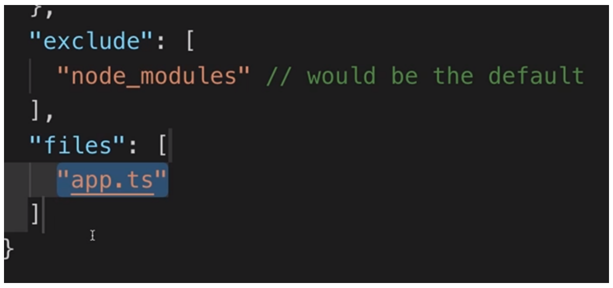


```
    /* Experimental Options */
    // "experimentalDecorators": true,
    // "emitDecoratorMetadata": true,
  },
  "exclude": [
    "node_modules" // would be the default
  ],
  "include": [
    "app.ts",
    "analytics.ts"
  ]
```

include and **exclude** are used to include specific files and exclude others.

basically, this will include the files/folders under include minus the files/folders in exclude.

if you only have include, it will automatically include all files minus the files listed in exclude

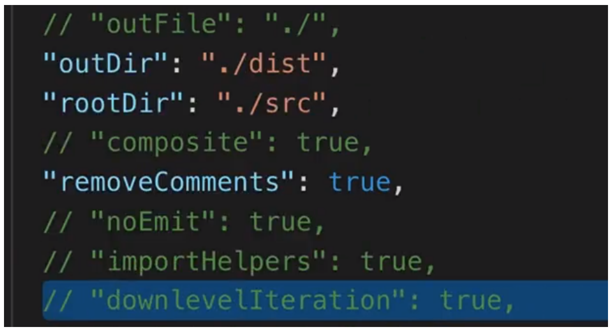


```
  },
  "exclude": [
    "node_modules" // would be the default
  ],
  "files": [
    "app.ts"
  ]
}
```

files is similar to include, except you can't specify entire folders, only files that you want to include. so if you're working on a smaller project, this is for you.

compilerOPTIONS

compiler options allow you to customize how .ts files are compiled by TS.



```
    // "outFile": "./",
    "outDir": "./dist",
    "rootDir": "./src",
    // "composite": true,
    "removeComments": true,
    // "noEmit": true,
    // "importHelpers": true,
    // "downlevelIteration": true,
```

outDir: tells TS where the compiled JS should be output to.

rootDir: tells TS where exactly it should start looking for .ts files. In this example, if there are ts files found outside of the src folder, they will not be compiled.

noEmit: does not emit outputs. compiler only checks if code has potential errors