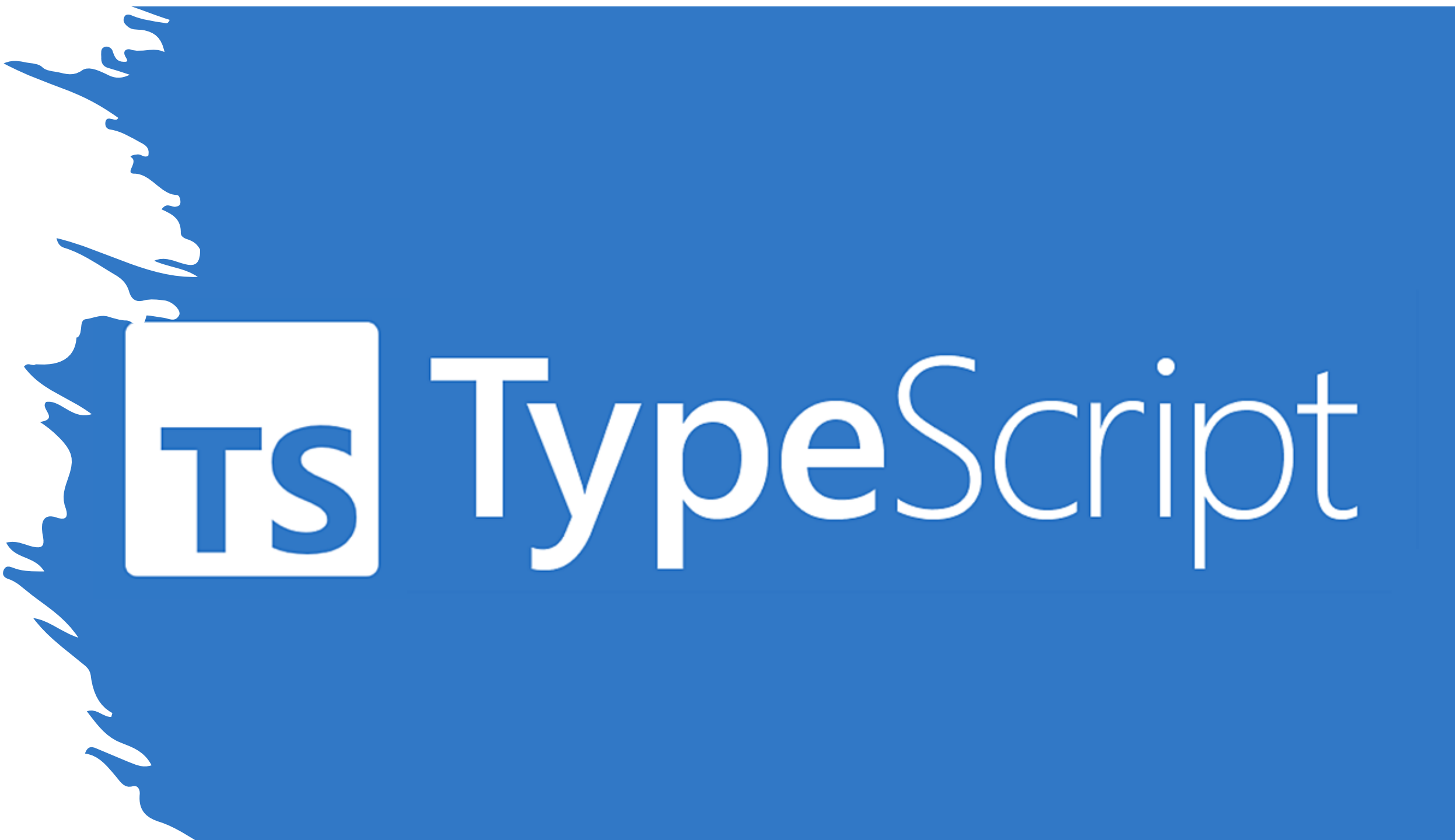

TS
TypeScript

Our Goal

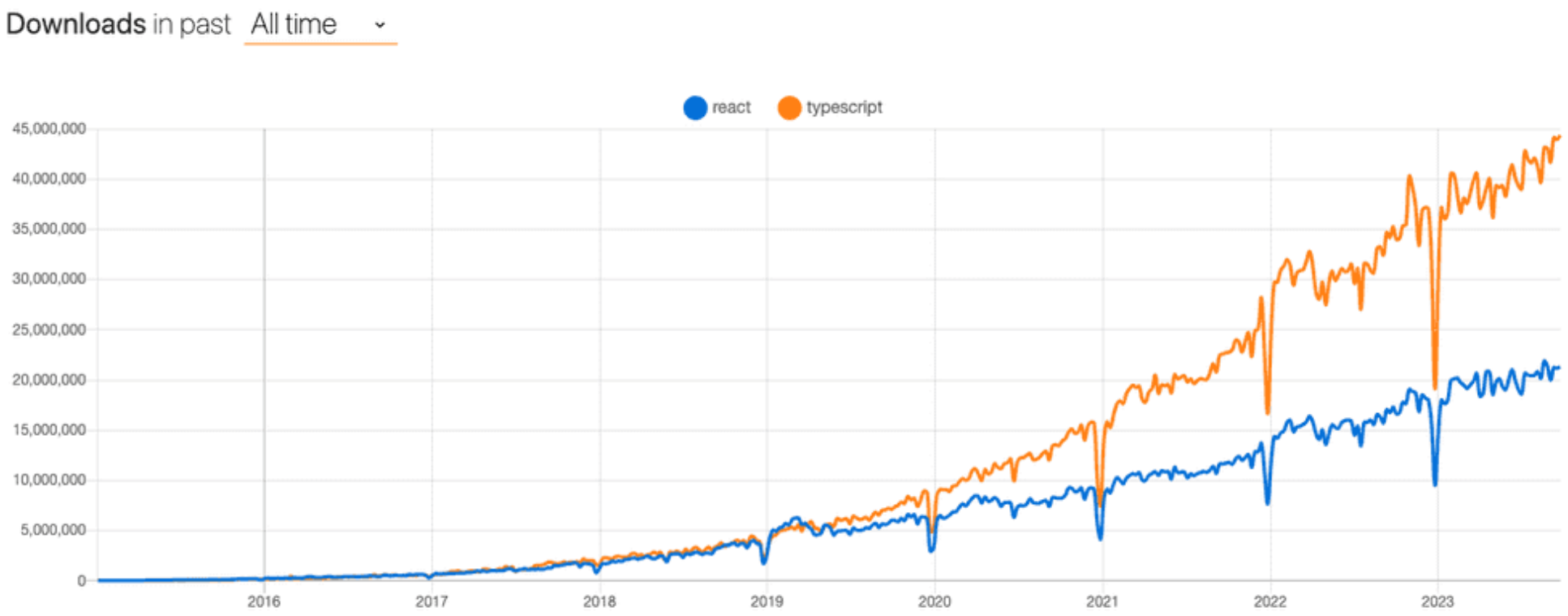
By the end, I want you to have a rock-solid mental model, that will allow you to design, discuss and understand types.

What is TypeScript?

- TypeScript is an open source, typed syntactic superset of JavaScript

- Compiles to clean, readable JS
- Three parts: Language, Language Server and Compiler
- Kind of like a fancy linter — helps purely at build time

TypeScript is *increasingly popular*



Why types are a big deal 😳

It allows developers to leave more of their intent **"on the page",** especially when it comes to articulating constraints.

This kind of intent is often missing from JS code. For example:

```
function add(a, b) {
  return a + b
}
```

Is this meant to take numbers as args? strings? both?

What if someone who interpreted **a** and **b** as numbers made this "backwards-compatible change?"

```
function add(a, b, c = 0) {
  return a + b + c
}
```

We're headed for trouble if we decide to pass strings in for **a** and **b**!

Types *make the author's intent more clear*

```
function add(a: number, b: number): number {
  return a + b
}
add(3, "4")
```

```
Argument of type 'string' is not assignable to parameter of type 'number'.
```

This code is more readable and self-documenting, in that it's *clearer how the author of this function intends it to be used*.

Why types are a big deal 😳

TypeScript moves some kinds of common programming errors from runtime to compile time

Examples:

- Potentially absent Values (null or undefined).
- Incomplete refactoring.
- Breakage around internal code contracts (e.g., an argument becomes required)

It serves as the foundation for a great code authoring experience

‘Hello World’ with TypeScript

TypeScript Setup using tsc

- pnpm install -g typescript

- tsc index.ts
- node index.js

1. Install Ts globally.
2. TypeScript compiler (tsc) will transpile your TypeScript code into JavaScript
3. Then Node index.js will run your compiled JavaScript file

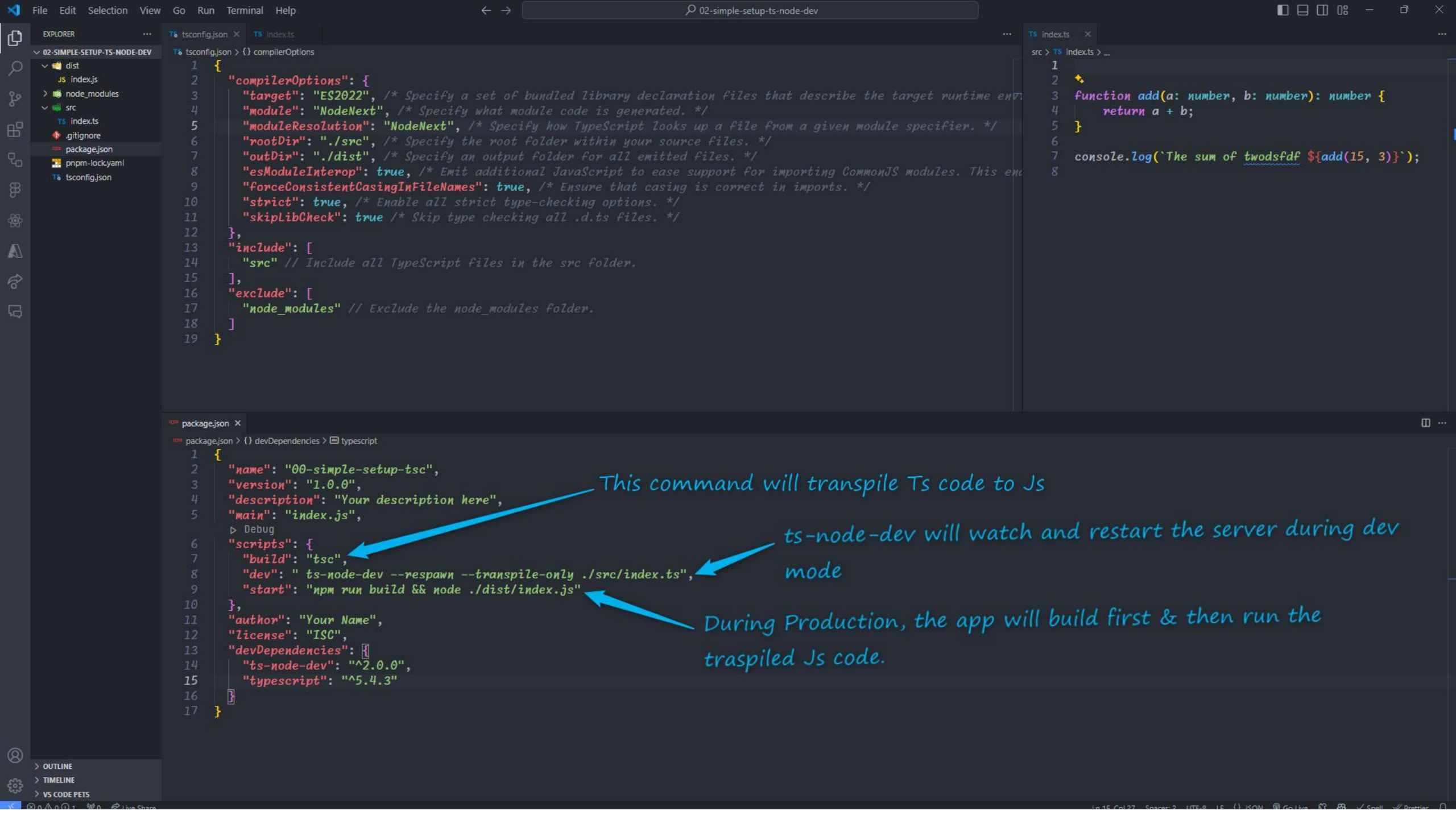
tsc VS ts-node-dev

Tsc (TypeScript Compiler)

- **Installation**: Comes bundled with the TypeScript package. No need to install separately.
- **Purpose**: The TypeScript compiler (tsc) is responsible for transpiling TypeScript files (.ts) into JavaScript files (.js). It processes all files according to your tsconfig.json configuration.
- **Compilation**: It compiles the entire project based on the specified settings in your tsconfig.json.
- **Output**: Produces JavaScript files that can be executed directly by Node.js.
- **Usage**:
  - Typically used for **production builds**.
  - Faster than ts-node.
  - Generates separate .js files for each .ts file.

ts-node

- **Installation**: pnpm install -g ts-node or pnpm install ts-node --save-dev.
- **Purpose**: Ts-node is a wrapper for Node.js's node executable. It installs a TypeScript-enabled module loader that compiles TypeScript code on-the-fly as needed.
- **Interpretation**: Instead of compiling all files upfront, ts-node interprets TypeScript code directly using a JavaScript interpreter.
- **Usage**:
  - Commonly used for **development purposes**.
  - Ideal for running in **watch mode** during development.
  - Convenient for quick iterations and debugging.
  - Acts as a REPL environment for TypeScript.
- **Performance**: Slower than tsc because it compiles TypeScript code on-the-fly.
- **Debugging**: Easier to debug because it preserves the original TypeScript code.
- Sample Usage:
  - **ts-node-dev --respawn** => **ts-node-dev --transpile-only** file.ts => to skip type-checking and only transpile ts code to js.
  - **ts-node --files** file.ts => to enable file watching and recompilation on file changes.


File Edit Selection View Go Run Terminal Help
02-simple-setup-ts-node-dev
EXPLORER
02-SIMPLE-SETUP-TS-NODE-DEV
dist
index.js
node_modules
src
index.ts
.gitignore
package.json
pnpm-lock.yaml
tsconfig.json
tsconfig.json
{} compilerOptions
{
"compilerOptions": {
"target": "ES2022", /* Specify a set of bundled library declaration files that describe the target runtime env
"module": "NodeNext", /* Specify what module code is generated. */
"moduleResolution": "NodeNext", /* Specify how TypeScript looks up a file from a given module specifier. */
"rootDir": "./src", /* Specify the root folder within your source files. */
"outDir": "./dist", /* Specify an output folder for all emitted files. */
"esModuleInterop": true, /* Emit additional JavaScript to ease support for importing CommonJS modules. This en
"forceConsistentCasingInFileNames": true, /* Ensure that casing is correct in imports. */
"strict": true, /* Enable all strict type-checking options. */
"skipLibCheck": true /* Skip type checking all .d.ts files. */
},
"include": [
"src" // Include all TypeScript files in the src folder.
],
"exclude": [
"node_modules" // Exclude the node_modules folder.
]
}
index.ts
src > index.ts > ...
function add(a: number, b: number): number {
return a + b;
}
console.log(`The sum of twodsfdf ${add(15, 3)}`);
package.json
package.json > {} devDependencies > typescript
{
"name": "00-simple-setup-tsc",
"version": "1.0.0",
"description": "Your description here",
"main": "index.js",
Debug
"scripts": {
"build": "tsc",
"dev": " ts-node-dev --respawn --transpile-only ./src/index.ts",
"start": "npm run build && node ./dist/index.js"
},
"author": "Your Name",
"license": "ISC",
"devDependencies": {
"ts-node-dev": "^2.0.0",
"typescript": "^5.4.3"
}
}
This command will transpile Ts code to Js
ts-node-dev will watch and restart the server during dev mode
During Production, the app will build first & then run the traspiled Js code.
OUTLINE
TIMELINE
VS CODE PETS

Variables and Values

Variable Declarations & Inference

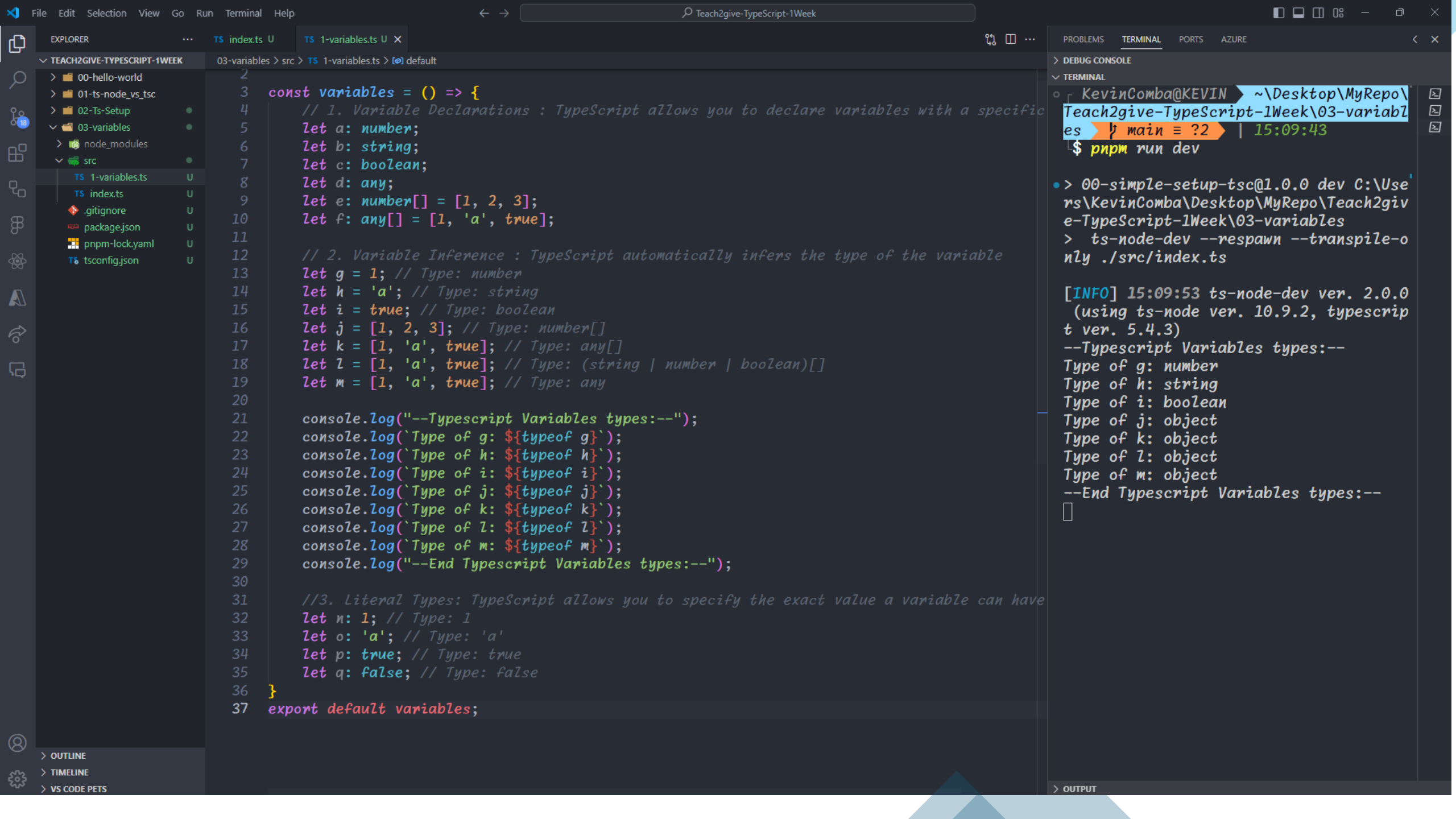
Variables

A variable is a named memory location that can hold a value.

Variables can store a wide range of data types, such as numbers, strings, and arrays.

A variable is declared by specifying its name, data type, and optionally an initial value.

Once a variable is declared, it can be ***read*** and ***updated*** in other parts of the program


File Edit Selection View Go Run Terminal Help
Teach2give-TypeScript-1Week
EXPLORER
index.ts U
1-variables.ts U
TEACH2GIVE-TYPESCRIPT-1WEEK
00-hello-world
01-ts-node_vs_tsc
02-Ts-Setup
03-variables
node_modules
src
1-variables.ts U
index.ts U
.gitignore U
package.json U
pnpm-lock.yaml U
tsconfig.json U
03-variables > src > 1-variables.ts > default
const variables = () => {
// 1. Variable Declarations : TypeScript allows you to declare variables with a specific
let a: number;
let b: string;
let c: boolean;
let d: any;
let e: number[] = [1, 2, 3];
let f: any[] = [1, 'a', true];
// 2. Variable Inference : TypeScript automatically infers the type of the variable
let g = 1; // Type: number
let h = 'a'; // Type: string
let i = true; // Type: boolean
let j = [1, 2, 3]; // Type: number[]
let k = [1, 'a', true]; // Type: any[]
let l = [1, 'a', true]; // Type: (string | number | boolean)[]
let m = [1, 'a', true]; // Type: any
console.log("--Typescript Variables types:--");
console.log(`Type of g: ${typeof g}`);
console.log(`Type of h: ${typeof h}`);
console.log(`Type of i: ${typeof i}`);
console.log(`Type of j: ${typeof j}`);
console.log(`Type of k: ${typeof k}`);
console.log(`Type of l: ${typeof l}`);
console.log(`Type of m: ${typeof m}`);
console.log("--End Typescript Variables types:--");
//3. Literal Types: TypeScript allows you to specify the exact value a variable can have
let n: 1; // Type: 1
let o: 'a'; // Type: 'a'
let p: true; // Type: true
let q: false; // Type: false
}
export default variables;
OUTLINE
TIMELINE
VS CODE PETS
PROBLEMS TERMINAL PORTS AZURE
DEBUG CONSOLE
TERMINAL
KevinComba@KEVIN ~\Desktop\MyRepo\Teach2give-TypeScript-1Week\03-variables main ≡ ?2 | 15:09:43
$ pnpm run dev
> 00-simple-setup-tsc@1.0.0 dev C:\Users\KevinComba\Desktop\MyRepo\Teach2give-TypeScript-1Week\03-variables
> ts-node-dev --respawn --transpile-only ./src/index.ts
[INFO] 15:09:53 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.9.2, typescript ver. 5.4.3)
--Typescript Variables types:--
Type of g: number
Type of h: string
Type of i: boolean
Type of j: object
Type of k: object
Type of l: object
Type of m: object
--End Typescript Variables types:--
OUTPUT

Special Types

TypeScript Special Types

Never

The never type represents the value that will never happen. We use it as the *return type of a function*, which does not return a value(infinite loop or function that always throws an error). For example, the function that always throws an exception is shown below

```
//Infinite loop
// Inferred return type: never
var x = function infiniteLoop() {
  while (true) {
  }
}

// Always throws an error
// Inferred return type: never
var y=function throwError() {
 throw new Error("Some errors occurred");
}
```

TypeScript Special Types

Void

Void represents the absence of any return value. For example, the following function prints “hello” and returns without returning a value. Hence the return type is void.

```
function sayHello(): void {
    console.log('Hello!')
}
```

TypeScript Special Types

Unknown

unknown type means that the type of variable is not known. It is the type-safe counterpart of any

```
let unknownVar:unknown;

unknownVar = true;                  //boolean
unknownVar = 10;                    //number
unknownVar = 10n;                   //BigInt  >>=ES2020
unknownVar = "Hello World";         //String
unknownVar = ["1","2","3","4"]      //Array
unknownVar = {firstName:'', lastName:''}; // Object
unknownVar = null;                  // null
unknownVar = undefined;             // undefined
unknownVar = Symbol("key");         // Symbol
```

But cannot assign unknown to any other types.

```
let value: unknown;

let value1: boolean = value;  // Error
let value2: number = value;   // Error
let value3: string = value;   // Error
let value4: object = value;   // Error
let value5: any[] = value;    // Error
let value6: Function = value; // Error
```

TypeScript Special Types

Null and Undefined

JavaScript has two ways to refer to the null. They are null and undefined and are two separate data types in Typescript as well. *The null and undefined are subtypes of all other types*. That means you can assign null and undefined to something like a *number*.

Undefined : Denotes value is given to all uninitialized variables.
Null : Represents the intentional absence of object value.

```
1
2 let u: undefined = undefined;
3 let n: null = null;
4
```

Objects

Objects

Variables

In general, object types are defined by:

- The **names** of the properties that are (or may be) present
- The **types** of those properties

```
/**
 * Print information about a car to the console
 * @param car - the car to print
 */
function printCar(car: {
  make: string
  model: string
  year: number
}) {
  console.log(`${car.m
```

```
make
model
```

Optional Properties

What if we take our car example a bit further by adding a fourth property that's only present sometimes?

Note that the type of chargeVoltage is now number | undefined

```
function printCar(car: {
  make: string
  model: string
  year: number
  chargeVoltage?: number
}) {
  let str = `${car.make} ${car.model} (${car.year})`
  car.chargeVoltage
```

```
(property) chargeVoltage?: number | undefined
```

```
  if (typeof car.chargeVoltage !== "undefined")
    str += `// ${car.chargeVoltage}v`
```

```
(property) chargeVoltage?: number
```

```
  console.log(str)
}
```

```ts
const ObjectsArraysTuples = () => {
    //  Objects
    const person: { name: string; age: number; } = { name: 'Max', age: 30 };    //CONST = Obj must  be initialized

    let car: {
        make: string
        model: string
        year: number
        chargeVoltage?: number     // Optional property (?) =  The type is number | undefined -not present unless car is electric.
    };

    // Initialize the object
    car = { make: 'Tesla', model: 'Model 3', year: 2020 }

    console.log(`--Ts Objects types:--`);
    console.log(`Type of person: ${typeof person}`);
    console.log(`Type of car: ${typeof car}`);
    console.log(`--------end-------`);

    console.log(`--Ts Objects property types:--`);
    console.log(`Type of car.make: ${typeof car.make}, car.model: ${typeof car.model}, car.year: ${typeof car.year}, car.chargeVoltage: ${typeof car.
    chargeVoltage}`);
    console.log(`Type of person.name: ${typeof person.name}, person.age: ${typeof person.age}`);
    console.log(`--------end-------`);

    console.log(`--Ts Objects property values:--`);
    console.log(`Value of car.make: ${car.make}, car.model: ${car.model}, car.year: ${car.year}`);
    console.log(`Value of person.name: ${person.name}, person.age: ${person.age}`);
    console.log(`--------end-------`);

}

export default ObjectsArraysTuples;
```

Arrays

Array Types

Describing types for arrays is often as easy as adding [] to the end of the array member's type. For example, the type for an array of strings would look like string[]. Later using various methods such as push() , splice() , and concat() . Arrays can be of a fixed length or dynamically resized as needed, and they can be used with various array methods to perform common operations like sorting, filtering, and mapping

```
// create an array
const numbers: number[] = [1, 2, 3];
let letters: string[] = ["a", "b", "c"];

// iterate over an array
for (let n of numbers) {
  // ...
}

// array method examples
numbers.pop();  // remove last item
const doubled = numbers.map(n => n * 2);  // double each number
```

Readonly Array

The readonly keyword can prevent arrays from being changed.

```
const names: readonly string[] = ["Dylan"];
names.push("Jack"); // Error: Property 'push' does not exist on type 'readonly string[]'.
// try removing the readonly modifier and see if it works?
```

Type Inference

TypeScript can infer the type of an array if it has values.

```
const numbers = [1, 2, 3]; // inferred to type number[]
numbers.push(4); // no error
// comment line below out to see the successful assignment
numbers.push("2"); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.
let head: number = numbers[0]; // no error
```

```ts

//Arrays
let colors: string[] = ['red', 'green', 'blue'];
let numbers: number[] = [1, 2, 3];
let truths: boolean[] = [true, false, true];

console.log(`--Ts Arrays types:--`);
console.log(`Type of colors: ${typeof colors}`);
console.log(`Type of numbers: ${typeof numbers}`);
console.log(`Type of truths: ${typeof truths}`);
console.log(`--------end-------`);

console.log(`--Ts Arrays property types:--`);
console.log(`Type of colors[0]: ${typeof colors[0]}, colors[1]: ${typeof colors[1]}, colors[2]: ${typeof colors[2]}`);
console.log(`Type of numbers[0]: ${typeof numbers[0]}, numbers[1]: ${typeof numbers[1]}, numbers[2]: ${typeof numbers[2]}`);
console.log(`Type of truths[0]: ${typeof truths[0]}, truths[1]: ${typeof truths[1]}, truths[2]: ${typeof truths[2]}`);

console.log(`--------end-------`);

//Arrays Operations

let colors2: string[] = ['red', 'green', 'blue'];
colors2.push('yellow');
colors2.push('purple');
colors2.pop();
console.log(`--Ts Arrays Operations:--`);
console.log(`Colors2: ${colors2}`);
console.log(`--------end-------`);

```

Tuples

Tuples

A tuple is a typed array with a pre-defined length and types for each index.

Let's imagine we define a convention where we can represent the same "2002 Toyota Corolla" as

```
//          [Year, Make,     Model    ]
let myCar = [2002, "Toyota", "Corolla"]
// destructured assignment is convenient here!
const [year, make, model] = myCar
```

TypeScript handles inference in this case

```
let myCar = [2002, "Toyota", "Corolla"]
    let myCar: (string | number)[]
const [year, make, model] = myCar
                  const model: string | number
```

| means "OR", so we can think of string | number means either a string or a number.

TypeScript has chosen the most specific type that describes the entire contents of the array. This is not quite what we wanted, in that:

- it allows us to break our convention where the year always comes first
- it doesn't quite help us with the "finite length" aspect of tuples

Explicitly state the type of a tuple

```
let myCar: [number, string, string] = [
  2002,
  "Toyota",
  "Corolla",
]
// ERROR: not the right convention
myCar = ["Honda", 2017, "Accord"]
```

```
Type 'string' is not assignable to type 'number'.
Type 'number' is not assignable to type 'string'.
```

```
// ERROR: too many items
myCar = [2017, "Honda", "Accord", "Sedan"]
```

```
Type '[number, string, string, string]' is not assignable to type '[number, string,
string]'.
  Source has 4 element(s) but target allows only 3.
```

```
const [year, make, model] = myCar
```

```
const year: number
```

```
make
```

```
const make: string
```

Now, we get errors in the places we expect, and all types work out as we hoped

Readonly Tuples

Tuples are just regular JS Arrays.

```
// Source
const numPair: [number, number] = [4, 5];
```

This imposes some degree of limitation on how tuples can be typed. For example, an Array allows new things to be .push(...)ed into them, allow .splice(...) and so on. At runtime these methods will exist on every tuple, and the types reflect that.

Typescript provides a lot of the support you'd hope for on assignment:

```
const numPair: [number, number] = [4, 5];
      const numPair: [number, number]
const numTriplet: [number, number, number] = [7];
Type '[number]' is not assignable to type '[number, number, number]'.
  Source has 1 element(s) but target requires 3.
```

and we see something interesting happening with .length

```
[101, 102, 103].length
                (property) Array<number>.length: number
numPair.length
          (property) length: 2
```

but we get no protection around push and pop, which effectively would change the type of the tuple

```
numPair.push(6) // [4, 5, 6]
numPair.pop() // [4, 5]
numPair.pop() // [4]
numPair.pop() // []

numPair.length  // ❌ DANGER ❌
          (property) length: 2
```

Readonly Tuples

If we are ok with treating this tuple as read-only, we can state so, and get a lot more safety around mutation.

```
const roNumPair: readonly [number, number] = [4, 5]
roNumPair.length
```

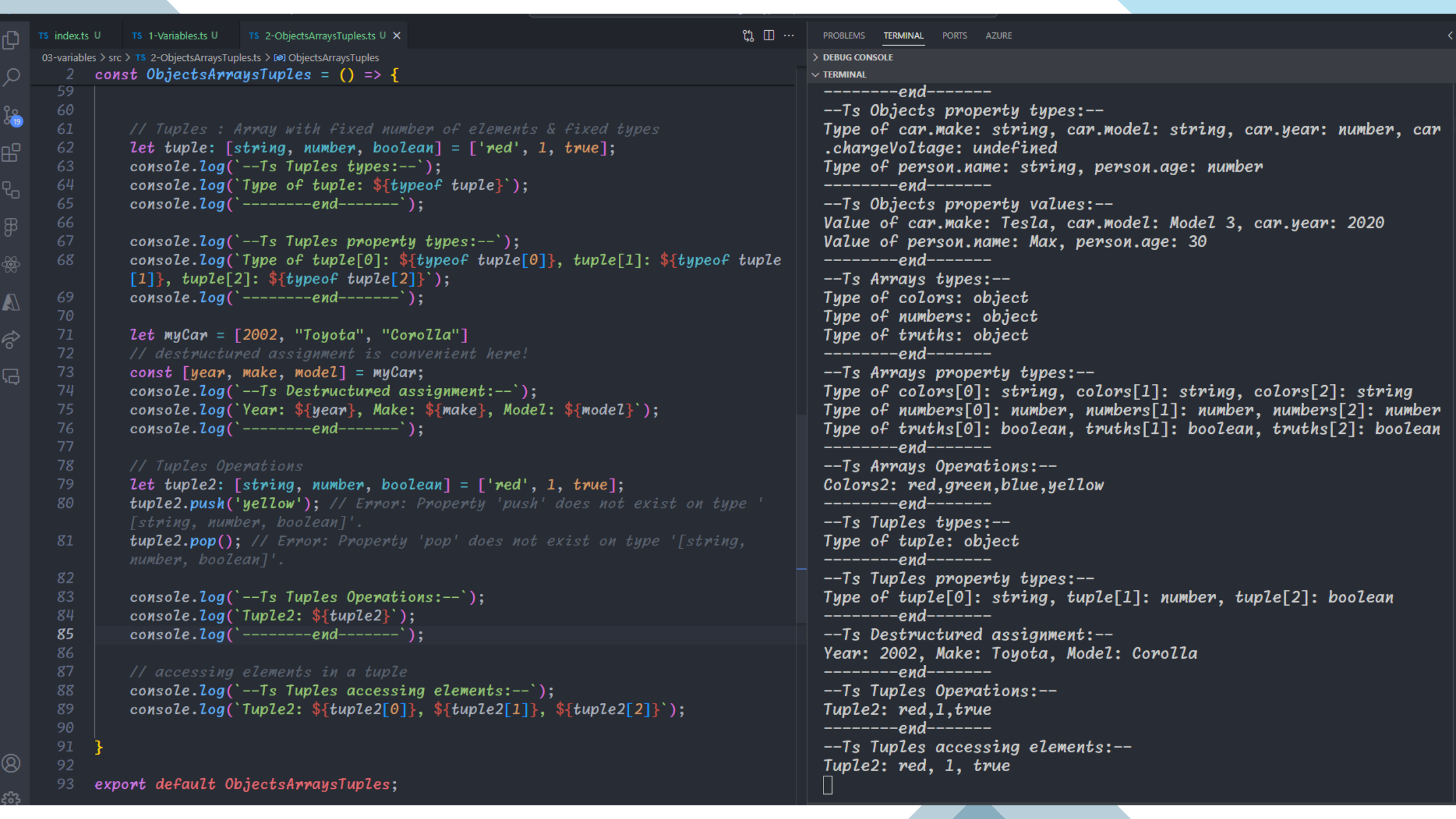
```
(property) length: 2
```

```
roNumPair.push(6) // [4, 5, 6]
```

```
Property 'push' does not exist on type 'readonly [number, number]'.
```

```
roNumPair.pop() // [4, 5]
```

```
Property 'pop' does not exist on type 'readonly [number, number]'.
```

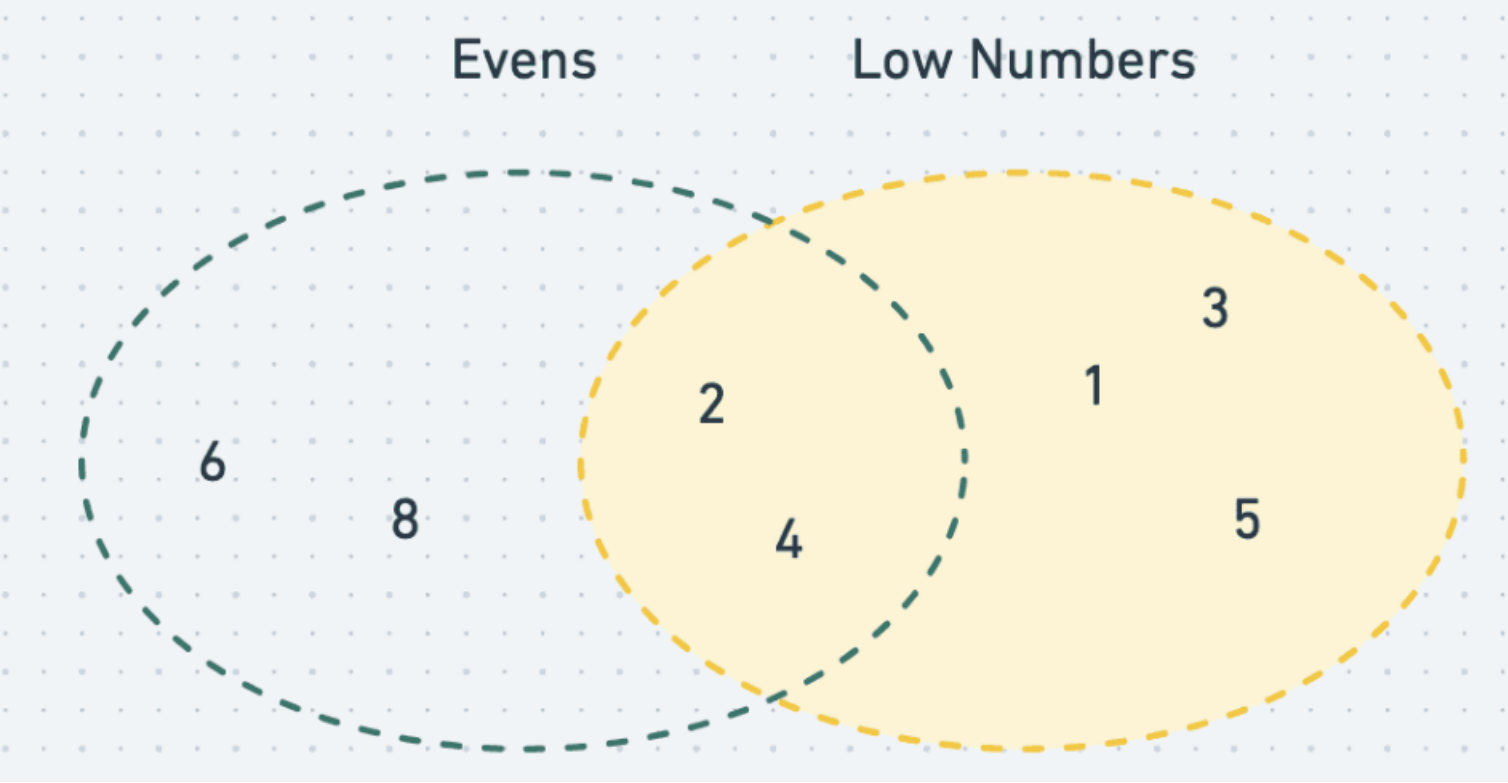

index.ts U
1-Variables.ts U
2-ObjectsArraysTuples.ts U
03-variables > src > 2-ObjectsArraysTuples.ts > ObjectsArraysTuples
const ObjectsArraysTuples = () => {
// Tuples : Array with fixed number of elements & fixed types
let tuple: [string, number, boolean] = ['red', 1, true];
console.log(`--Ts Tuples types:--`);
console.log(`Type of tuple: ${typeof tuple}`);
console.log(`--------end-------`);
console.log(`--Ts Tuples property types:--`);
console.log(`Type of tuple[0]: ${typeof tuple[0]}, tuple[1]: ${typeof tuple[1]}, tuple[2]: ${typeof tuple[2]}`);
console.log(`--------end-------`);
let myCar = [2002, "Toyota", "Corolla"]
// destructured assignment is convenient here!
const [year, make, model] = myCar;
console.log(`--Ts Destructured assignment:--`);
console.log(`Year: ${year}, Make: ${make}, Model: ${model}`);
console.log(`--------end-------`);
// Tuples Operations
let tuple2: [string, number, boolean] = ['red', 1, true];
tuple2.push('yellow'); // Error: Property 'push' does not exist on type '[string, number, boolean]'.
tuple2.pop(); // Error: Property 'pop' does not exist on type '[string, number, boolean]'.
console.log(`--Ts Tuples Operations:--`);
console.log(`Tuple2: ${tuple2}`);
console.log(`--------end-------`);
// accessing elements in a tuple
console.log(`--Ts Tuples accessing elements:--`);
console.log(`Tuple2: ${tuple2[0]}, ${tuple2[1]}, ${tuple2[2]}`);
}
export default ObjectsArraysTuples;
PROBLEMS
TERMINAL
PORTS
AZURE
DEBUG CONSOLE
TERMINAL
--------end-------
--Ts Objects property types:--
Type of car.make: string, car.model: string, car.year: number, car.chargeVoltage: undefined
Type of person.name: string, person.age: number
--------end-------
--Ts Objects property values:--
Value of car.make: Tesla, car.model: Model 3, car.year: 2020
Value of person.name: Max, person.age: 30
--------end-------
--Ts Arrays types:--
Type of colors: object
Type of numbers: object
Type of truths: object
--------end-------
--Ts Arrays property types:--
Type of colors[0]: string, colors[1]: string, colors[2]: string
Type of numbers[0]: number, numbers[1]: number, numbers[2]: number
Type of truths[0]: boolean, truths[1]: boolean, truths[2]: boolean
--------end-------
--Ts Arrays Operations:--
Colors2: red,green,blue,yellow
--------end-------
--Ts Tuples types:--
Type of tuple: object
--------end-------
--Ts Tuples property types:--
Type of tuple[0]: string, tuple[1]: number, tuple[2]: boolean
--------end-------
--Ts Destructured assignment:--
Year: 2002, Make: Toyota, Model: Corolla
--------end-------
--Ts Tuples Operations:--
Tuple2: red,1,true
--------end-------
--Ts Tuples accessing elements:--
Tuple2: red, 1, true

Union and Intersection Types

Union and Intersection Types, Conceptually

Union and intersection types can conceptually be thought of as logical boolean operators (AND, OR) as they pertain to types. Here are a couple of example sets we'll use for this discussion

```
Evens = { 2, 4, 6, 8 }
OneThroughFive = { 1, 2, 3, 4, 5 }
```



Union Types |

A union type can be thought of as OR, for types, and TypeScript uses the pipe (|) symbol to represent the Union type operator

Using the example above, if we wanted to find OneThroughFive | Evens we'd combine all the members of the OneThroughFive set and all of the members of the Evens set.

```
OneThroughFive | Evens => { 1, 2, 3, 4, 5, 6, 8 }
```

If you think about the assumptions we could make about a member of this set at random, we couldn't be sure whether it's between 1 and 5, and we couldn't be sure whether it's odd.

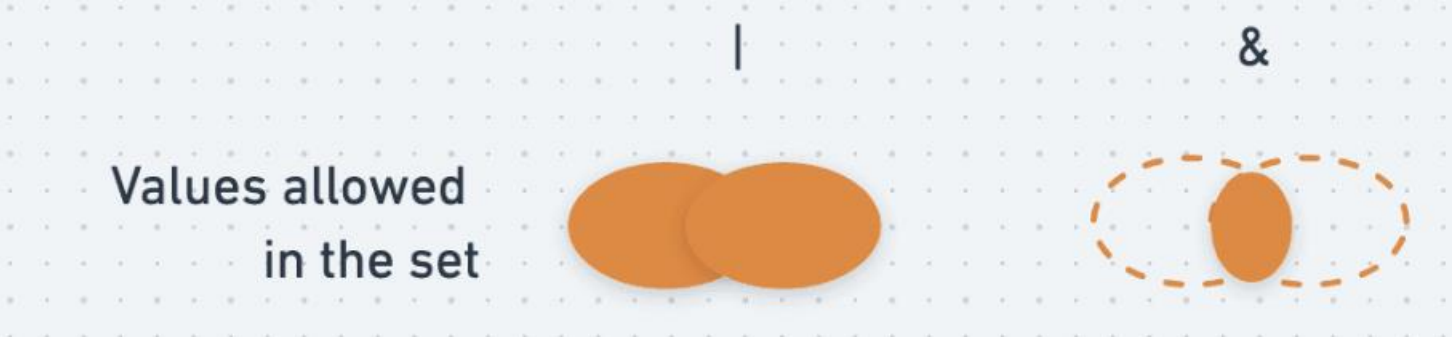
Intersection types &

An intersection type can be thought of as AND, for types, and TypeScript uses the ampersand (&) symbol to represent the Intersection type operator

Using the example again, if we wanted to find OneThroughFive & Evens we'd find all members that the OneThroughFive and Evens sets have in common

```
OneThroughFive & Evens => { 2, 4 }
```

If you think about the assumptions we could make about a member of this set at random, we couldn't be sure whether it's between 1 and 5, and we couldn't be sure whether it's odd.



Union Types in TypeScript

If we wanted to create a union type that represented the set { 1, 2, 3, 4, 5 } we could do it using the | operator. We can also use the type keyword to give this type a name.

```
type OneThroughFive = 1 | 2 | 3 | 4 | 5
```

```
type OneThroughFive = 1 | 2 | 3 | 4 | 5
```

```
let lowNumber: OneThroughFive = 3
```

```
let lowNumber: OneThroughFive
```

```
lowNumber = 8
```

```
Type '8' is not assignable to type 'OneThroughFive'.
```

and we could create another type called Evens to represent the set { 2, 4, 6, 8 }

```
type Evens = 2 | 4 | 6 | 8
```

```
type Evens = 2 | 4 | 6 | 8
```

```
let evenNumber: Evens = 2;
```

```
let evenNumber: Evens
```

```
evenNumber = 5;
```

```
Type '5' is not assignable to type 'Evens'.
```

Explicitly creating the union type is now simple

```
let evenOrLowNumber = 5 as Evens | OneThroughFive;
```

```
let evenOrLowNumber: 2 | 4 | 6 | 8 | 1 | 3 | 5
```

Intersection Types in TypeScript

If we wanted to create a union type that represented the set { 1, 2, 3, 4, 5 } we could do it using the | operator. We can also use the type keyword to give this type a name.

```
type OneThroughFive = 1 | 2 | 3 | 4 | 5
```

```
type OneThroughFive = 1 | 2 | 3 | 4 | 5
```

```
let lowNumber: OneThroughFive = 3
```

```
let lowNumber: OneThroughFive
```

```
lowNumber = 8
```

```
Type '8' is not assignable to type 'OneThroughFive'.
```

and we could create another type called Evens to represent the set { 2, 4, 6, 8 }

```
type Evens = 2 | 4 | 6 | 8
```

```
type Evens = 2 | 4 | 6 | 8
```

```
let evenNumber: Evens = 2;
```

```
let evenNumber: Evens
```

```
evenNumber = 5;
```

```
Type '5' is not assignable to type 'Evens'.
```

Explicitly creating the union type is now simple

```
let evenOrLowNumber = 5 as Evens | OneThroughFive;
```

```
let evenOrLowNumber: 2 | 4 | 6 | 8 | 1 | 3 | 5
```

index.ts U | 3-Union-Intersection.ts U

03-variables > src > 3-Union-Intersection.ts > UnionAndIntersection

```ts
const UnionAndIntersection = () => {
    // Union Types in TypeScript
    let unionType: number | string;
    unionType = 1; // OK
    console.log(`Union Type: ${unionType}`);
    console.log(`Type of unionType: ${typeof unionType}`);
    unionType = 'Hello'; // OK
    console.log(`Union Type: ${unionType}`);
    console.log(`Type of unionType: ${typeof unionType}`);
    // unionType = true; // Error: Type 'boolean' is not assignable to type 'string | number'.
    console.log(`--------end-------`);

    // Intersection Types in TypeScript
    // Intersection basic types
    type A = { a: number };
    type B = { b: string };
    type C = { c: boolean };
    type ABC = A & B & C;
    let abc: ABC = { a: 1, b: 'Hello', c: true }; // OK
    console.log(`Intersection Type: ${abc}`);
    console.log(`Type of abc: ${typeof abc}`);
    // let abc3: ABC = { a: 1, b: 'Hello' }; // Error: Property 'c' is missing in type '{ a: number; b: string; }' but requi
    // console.log(`Intersection Type: ${abc3}`);
    console.log(`--------end-------`);

    // Intersection complex types
    interface Colorful {
        color: string;
    }
    interface Circle {
        radius: number;
    }
    type ColorfulCircle = Colorful & Circle;
    let colorfulCircle: ColorfulCircle = { color: 'red', radius: 42 }; // OK
    console.log(`Intersection Type: ${colorfulCircle}`);
    console.log(`Type of colorfulCircle: ${typeof colorfulCircle}`);
    console.log(`--------end-------`);

}
```

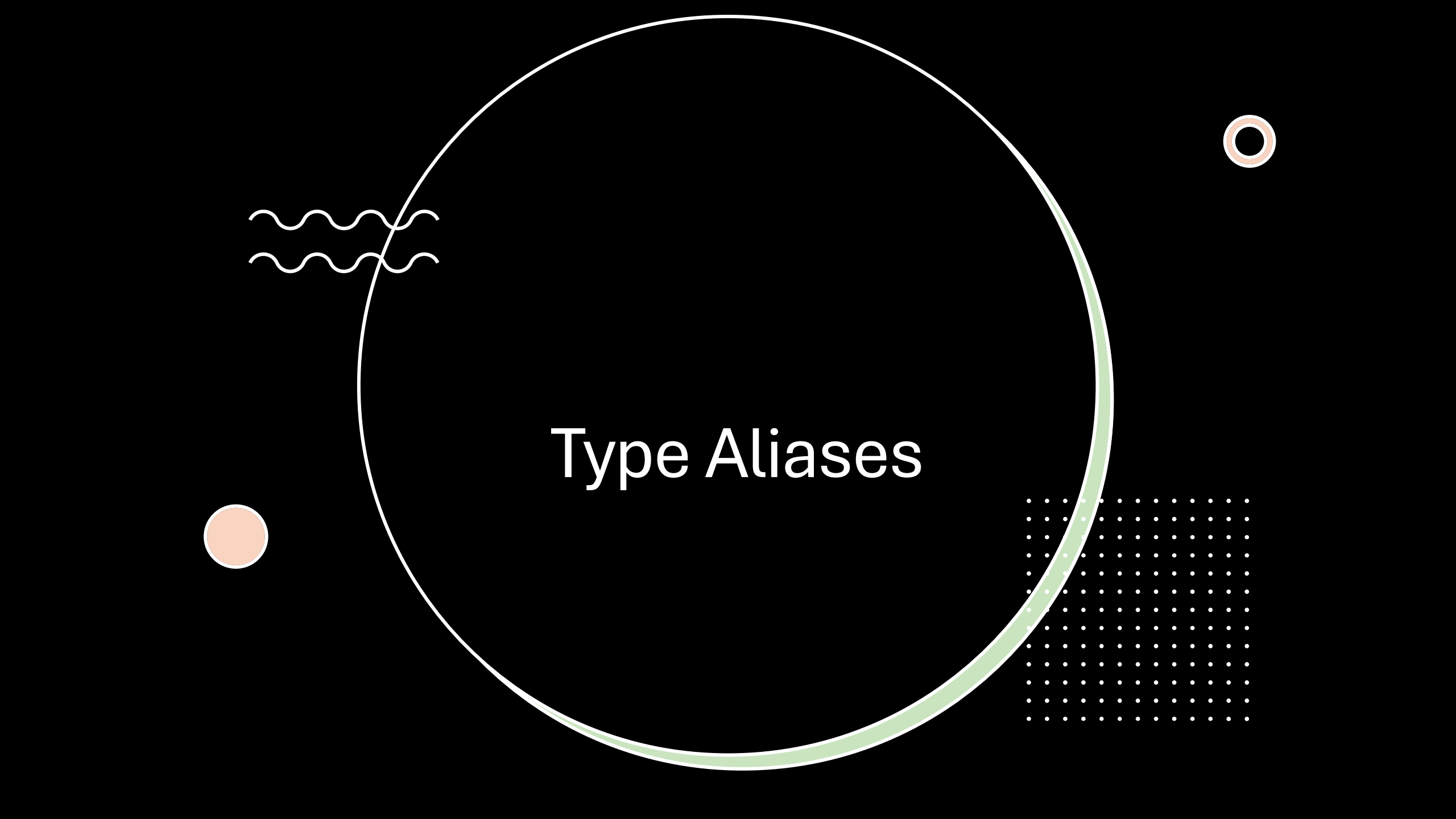
PROBLEMS | TERMINAL | PORTS | AZURE

DEBUG CONSOLE

TERMINAL

```
--------end-------
Intersection Type: [object Object]
Type of abc: object
Intersection Type: [object Object]
--------end-------
Intersection Type: [object Object]
Type of colorfulCircle: object
--------end-------
[INFO] 19:08:08 Restarting: C:\Users\KevinComba\Desktop\MyRepo\Teach2give-TypeScript-1Week\03-variables\src\3-Union-Intersection.ts has been modified
Union Type: 1
Type of unionType: number
Union Type: Hello
Type of unionType: string
--------end-------
Intersection Type: [object Object]
Type of abc: object
--------end-------
Intersection Type: [object Object]
Type of colorfulCircle: object
--------end-------
[INFO] 19:08:18 Restarting: C:\Users\KevinComba\Desktop\MyRepo\Teach2give-TypeScript-1Week\03-variables\src\index.ts has been modified
```

OUTPUT


Type Aliases

Interfaces and Type Aliases

TypeScript provides two mechanisms for centrally defining types and giving them useful and meaningful names:

Type Aliases

Type Aliases allow defining types with a custom name (an Alias). Type Aliases can be used for primitives like string or more complex types such as objects and arrays:

```
type CarYear = number
type CarType = string
type CarModel = string
type Car = {
  year: CarYear,
  type: CarType,
  model: CarModel
}

const carYear: CarYear = 2001
const carType: CarType = "Toyota"
const carModel: CarModel = "Corolla"
const car: Car = {
  year: carYear,
  type: carType,
  model: carModel
};
```

Type Aliases Benefits

- Type aliases help to address this, by allowing us to:

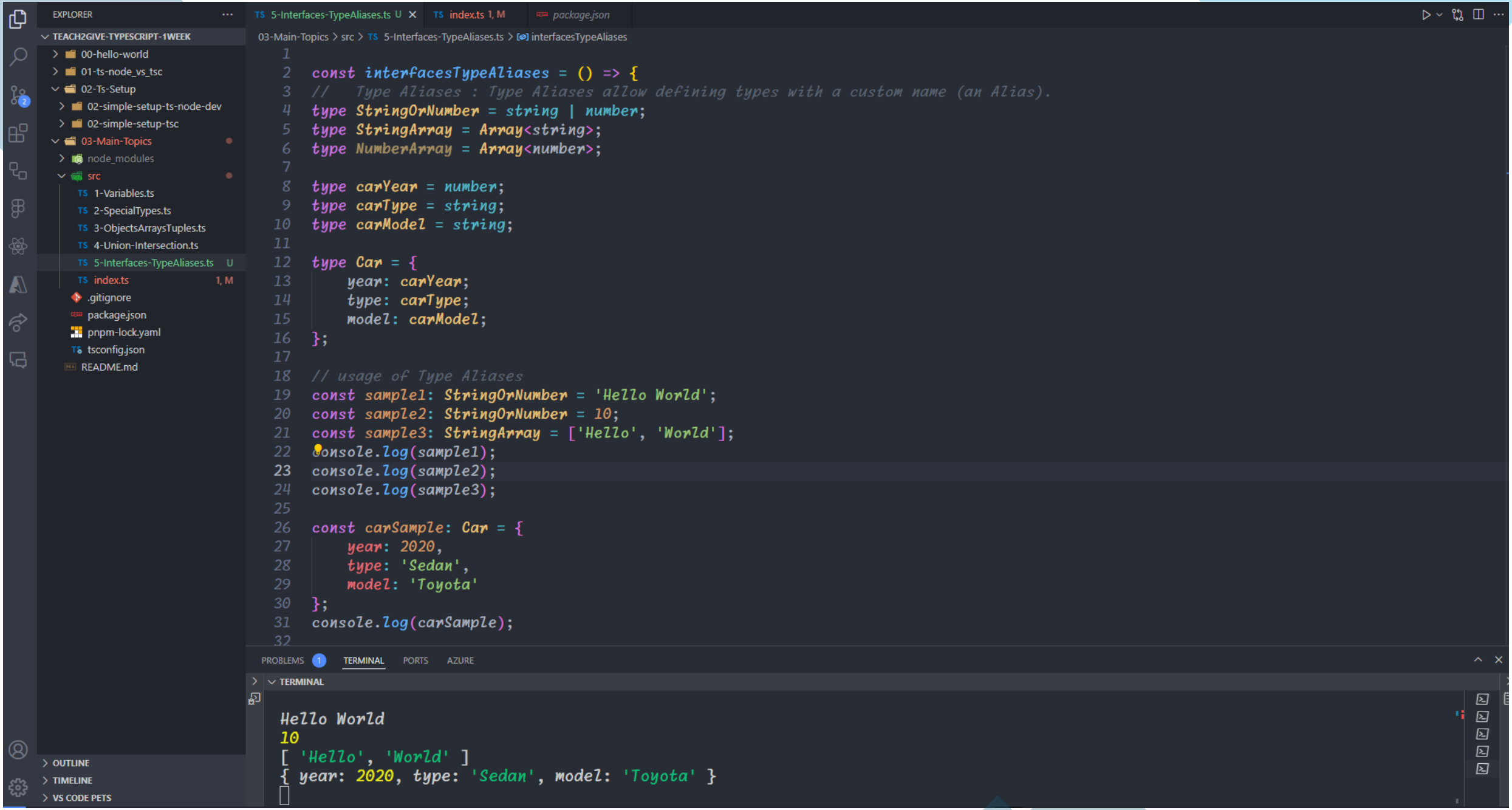
  - define **a more meaningful name** for this type
  - declare the shape of the type **in a single place**
  - **import and export** this type from modules, the same as if it were an importable/exportable value

```
//////////////////////////////////////////////////////////////
// @filename: types.ts
export type Amount = {
  currency: string
  value: number
}
//////////////////////////////////////////////////////////////
// @filename: utilities.ts
import { Amount } from "./types"

(alias) type Amount = {
    currency: string;
    value: number;
}
import Amount

function printAmount(amt: Amount) {
  console.log(amt)

(parameter) amt: Amount

  const { currency, value } = amt
  console.log(`${currency} ${value}`)

const currency: string

}
```

A few things to point out here: 😐

- This is a rare occasion where we see type information on the right-hand side of the assignment operator (=)
- We're using TitleCase to format the alias' name. This is a common convention
- As we can see below, we can only declare an alias of a given name once within a given scope. This is kind of like how a let or const variable declaration works

```
type Amount = {
Duplicate identifier 'Amount'.
  currency: string
  value: number
}

type Amount = {
Duplicate identifier 'Amount'.
  fail: "this will not work"
}
```


EXPLORER
TEACH2GIVE-TYPESCRIPT-1WEEK
00-hello-world
01-ts-node_vs_tsc
02-Ts-Setup
02-simple-setup-ts-node-dev
02-simple-setup-tsc
03-Main-Topics
node_modules
src
1-Variables.ts
2-SpecialTypes.ts
3-ObjectsArraysTuples.ts
4-Union-Intersection.ts
5-Interfaces-TypeAliases.ts U
index.ts 1, M
.gitignore
package.json
pnpm-lock.yaml
tsconfig.json
README.md
OUTLINE
TIMELINE
VS CODE PETS
5-Interfaces-TypeAliases.ts U
index.ts 1, M
package.json
03-Main-Topics > src > 5-Interfaces-TypeAliases.ts > interfacesTypeAliases
const interfacesTypeAliases = () => {
// Type Aliases : Type Aliases allow defining types with a custom name (an Alias).
type StringOrNumber = string | number;
type StringArray = Array<string>;
type NumberArray = Array<number>;
type carYear = number;
type carType = string;
type carModel = string;
type Car = {
year: carYear;
type: carType;
model: carModel;
};
// usage of Type Aliases
const sample1: StringOrNumber = 'Hello World';
const sample2: StringOrNumber = 10;
const sample3: StringArray = ['Hello', 'World'];
console.log(sample1);
console.log(sample2);
console.log(sample3);
const carSample: Car = {
year: 2020,
type: 'Sedan',
model: 'Toyota'
};
console.log(carSample);
PROBLEMS 1
TERMINAL
PORTS
AZURE
TERMINAL
Hello World
10
['Hello', 'World']
{ year: 2020, type: 'Sedan', model: 'Toyota' }

Inheritance in type aliases

You can create type aliases that combine existing types with new behavior by using Intersection (&) types.

```
type SpecialDate = Date & { getDescription(): string }

const newYearsEve: SpecialDate = Object.assign(
    new Date(),
    { getDescription: () => "Last day of the year" }
  )
newYearsEve.getD
```


EXPLORER
TEACH2GIVE-TYPESCRIPT-1WEEK
00-hello-world
01-ts-node_vs_tsc
02-Ts-Setup
02-simple-setup-ts-node-dev
02-simple-setup-tsc
03-Main-Topics
node_modules
src
1-Variables.ts
2-SpecialTypes.ts
3-ObjectsArraysTuples.ts
4-Union-Intersection.ts
5-Interfaces-TypeAliases.ts U
index.ts 1, M
.gitignore
package.json
pnpm-lock.yaml
tsconfig.json
README.md
OUTLINE
TIMELINE
VS CODE PETS
5-Interfaces-TypeAliases.ts U
index.ts 1, M
package.json
03-Main-Topics > src > 5-Interfaces-TypeAliases.ts > interfacesTypeAliases
const interfacesTypeAliases = () => {
};
// usage of Type Aliases
const sample1: StringOrNumber = 'Hello World';
const sample2: StringOrNumber = 10;
const sample3: StringArray = ['Hello', 'World'];
// console.log(sample1);
// console.log(sample2);
// console.log(sample3);
const carSample: Car = {
year: 2020,
type: 'Sedan',
model: 'Toyota'
};
console.log(carSample);
// extending Type Aliases
type Car2 = Car & { color: string };
const carSample2: Car2 = {
year: 2020,
type: 'Sedan',
model: 'Toyota',
color: 'Red'
};
console.log(carSample2);
}
PROBLEMS 1
TERMINAL
PORTS
AZURE
TERMINAL
[INFO] 16:43:23 Restarting: C:\Users\KevinComba\Desktop\MyRepo\Teach2give-TypeScript-1Week\03-Main-Topics\src\5-Interfaces-TypeAliases.ts has been modified
{ year: 2020, type: 'Sedan', model: 'Toyota' }
{ year: 2020, type: 'Sedan', model: 'Toyota', color: 'Red' }

Interfaces

Interfaces

Interfaces are similar to type aliases; except they only apply to object types.

```
{
  field: "value
}
```

```
interface Rectangle {
  height: number,
  width: number
}

const rectangle: Rectangle = {
  height: 20,
  width: 10
};
```

Extending Interfaces

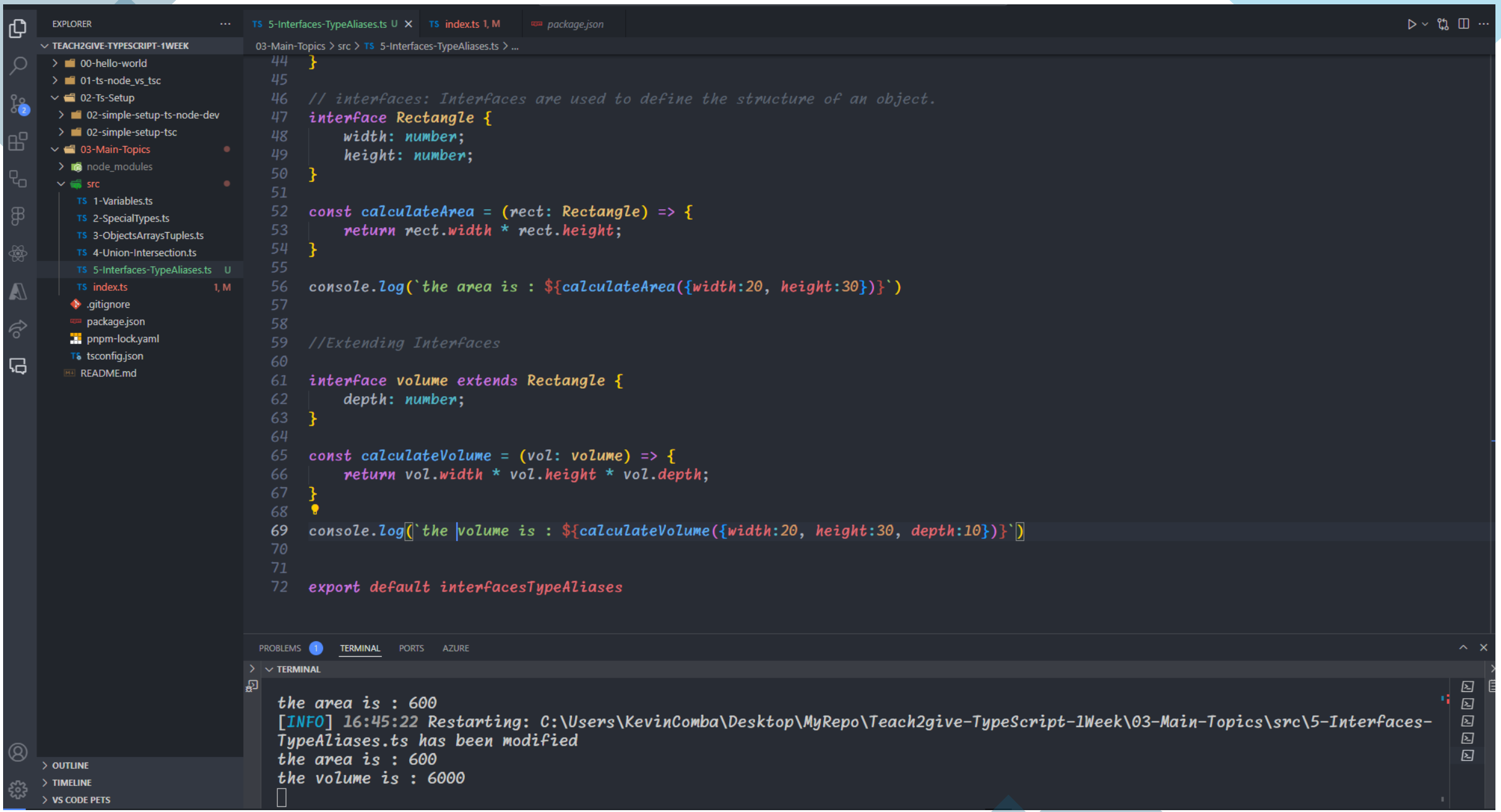
Extending an interface means you are creating a new interface with the same properties as the original, plus something new.

```
interface Rectangle {
  height: number,
  width: number
}

interface ColoredRectangle extends Rectangle {
  color: string
}

const coloredRectangle: ColoredRectangle = {
  height: 20,
  width: 10,
  color: "red"
};
```

Choosing whether to use type or interface

In many situations, either a type alias or an interface would be perfectly fine, however...

1. **If you need to define something other than an object type** (e.g., use of the | union type operator), you must use a type alias
2. If you need to define a type **to use with the implement's heritage term on a class**, use an interface
3. If you need to **allow consumers of your types to augment them**, you must use an interface.


EXPLORER
TEACH2GIVE-TYPESCRIPT-1WEEK
00-hello-world
01-ts-node_vs_tsc
02-Ts-Setup
02-simple-setup-ts-node-dev
02-simple-setup-tsc
03-Main-Topics
node_modules
src
1-Variables.ts
2-SpecialTypes.ts
3-ObjectsArraysTuples.ts
4-Union-Intersection.ts
5-Interfaces-TypeAliases.ts U
index.ts 1, M
.gitignore
package.json
pnpm-lock.yaml
tsconfig.json
README.md
OUTLINE
TIMELINE
VS CODE PETS
5-Interfaces-TypeAliases.ts U
index.ts 1, M
package.json
03-Main-Topics > src > 5-Interfaces-TypeAliases.ts > ...
44 }
45
46 // interfaces: Interfaces are used to define the structure of an object.
47 interface Rectangle {
48 width: number;
49 height: number;
50 }
51
52 const calculateArea = (rect: Rectangle) => {
53 return rect.width * rect.height;
54 }
55
56 console.log(`the area is : ${calculateArea({width:20, height:30})}`)
57
58
59 //Extending Interfaces
60
61 interface volume extends Rectangle {
62 depth: number;
63 }
64
65 const calculateVolume = (vol: volume) => {
66 return vol.width * vol.height * vol.depth;
67 }
68
69 console.log(`the volume is : ${calculateVolume({width:20, height:30, depth:10})}`)
70
71
72 export default interfacesTypeAliases
PROBLEMS 1
TERMINAL
PORTS
AZURE
TERMINAL
the area is : 600
[INFO] 16:45:22 Restarting: C:\Users\KevinComba\Desktop\MyRepo\Teach2give-TypeScript-1Week\03-Main-Topics\src\5-Interfaces-TypeAliases.ts has been modified
the area is : 600
the volume is : 6000

Functions

TypeScript Functions

TypeScript has a specific syntax for typing function parameters and return values.

Return Type

The type of the value returned by the function can be explicitly defined.

```
// the `: number` here specifies that this function returns a number
function getTime(): number {
  return new Date().getTime();
}
```

Void Return Type

The type void can be used to indicate a function doesn't return any value.

```
function printHello(): void {
  console.log('Hello!');
}
```

Parameters

unction parameters are typed with a similar syntax as variable declarations.

```
function multiply(a: number, b: number) {
  return a * b;
}
```

TypeScript Functions

Optional Parameters

By default TypeScript will assume all parameters are required, but they can be explicitly marked as optional.

```
// the `?` operator here marks parameter `c` as optional
function add(a: number, b: number, c?: number) {
  return a + b + (c || 0);
}
```

Default Parameters

For parameters with default values, the default value goes after the type annotation:

```
function pow(value: number, exponent: number = 10) {
  return value ** exponent;
}
```

Rest Parameters

Rest parameters can be typed like normal parameters, but the type must be an array as rest parameters are always arrays.

```
function add(a: number, b: number, ...rest: number[]) {
  return a + b + rest.reduce((p, c) => p + c, 0);
}
```


EXPLORER
TEACH2GIVE-TYPESCRIPT-1WEEK
00-hello-world
01-ts-node_vs_tsc
02-Ts-Setup
02-simple-setup-ts-node-dev
02-simple-setup-tsc
03-Main-Topics
node_modules
src
1-Variables.ts
2-SpecialTypes.ts
3-ObjectsArraysTuples.ts
4-Union-Intersection.ts
5-Interfaces-TypeAliases.ts U
6-Fuctions.ts U
index.ts 1, M
.gitignore
package.json
pnpm-lock.yaml
tsconfig.json
README.md
OUTLINE
TIMELINE
VS CODE PETS
5-Interfaces-TypeAliases.ts U
6-Fuctions.ts U
index.ts 1, M
package.json
03-Main-Topics > src > 6-Fuctions.ts > tsFuctions
const tsFuctions = () => {
//Arrow Function
const add = (a: number, b: number) => a + b;
// console.log(add(10, 20));
const perimeter = (a: number, b: number) => {
return 2 * (a + b);
}
console.log(`The perimeter is: ${perimeter(10, 20)}`);
//Optional Parameters
const fullName = (firstName: string, lastName?: string) => {
return lastName ? `${firstName} ${lastName}` : firstName;
}
console.log(`The full name is: ${fullName('John')}`);
console.log(`The full name is: ${fullName('John', 'Doe')}`);
//Default Parameters
const getSalary = (basic: number, bonus: number = 0) => {
return basic + bonus;
}
console.log(`The salary is: ${getSalary(1000)}`);
//Void Function
const greet = (): void => {
console.log('Hello World');
}
greet();
// rest parameters
const addNumbers = (...numbers: number[]) => {
return numbers.reduce((sum, num) => sum + num, 0);
}
console.log(`The sum is: ${addNumbers(10, 20, 30, 40)}`);
}
export default tsFuctions;

Callbacks

Callbacks

TypeScript, defining callback functions involves using **function signatures as types**. This powerful feature allows developers to create structured, type-safe callback mechanisms.

Basic Callback Type

To define a callback type, use the following syntax:

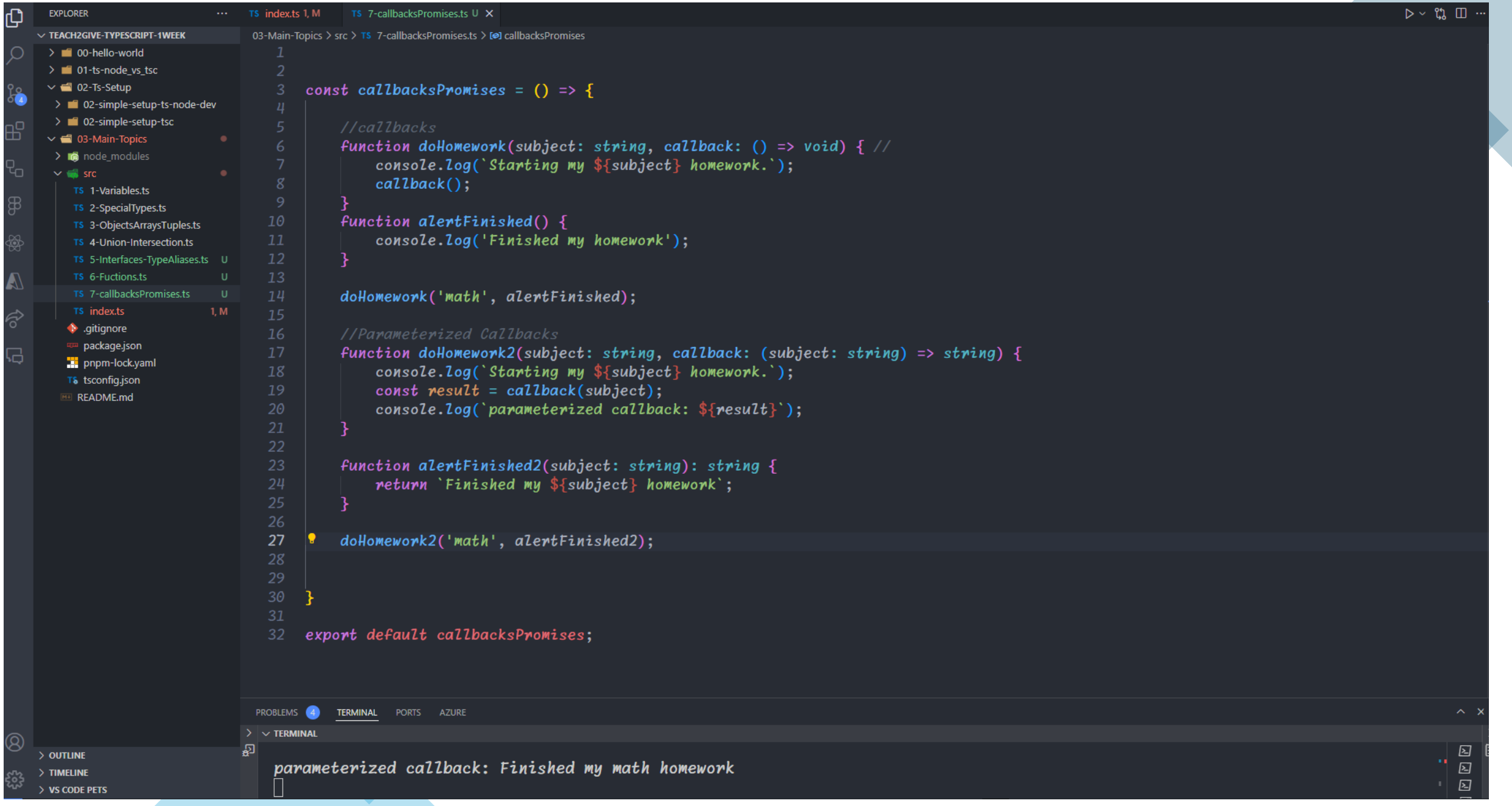
```
type MyCallback = () => void;
```

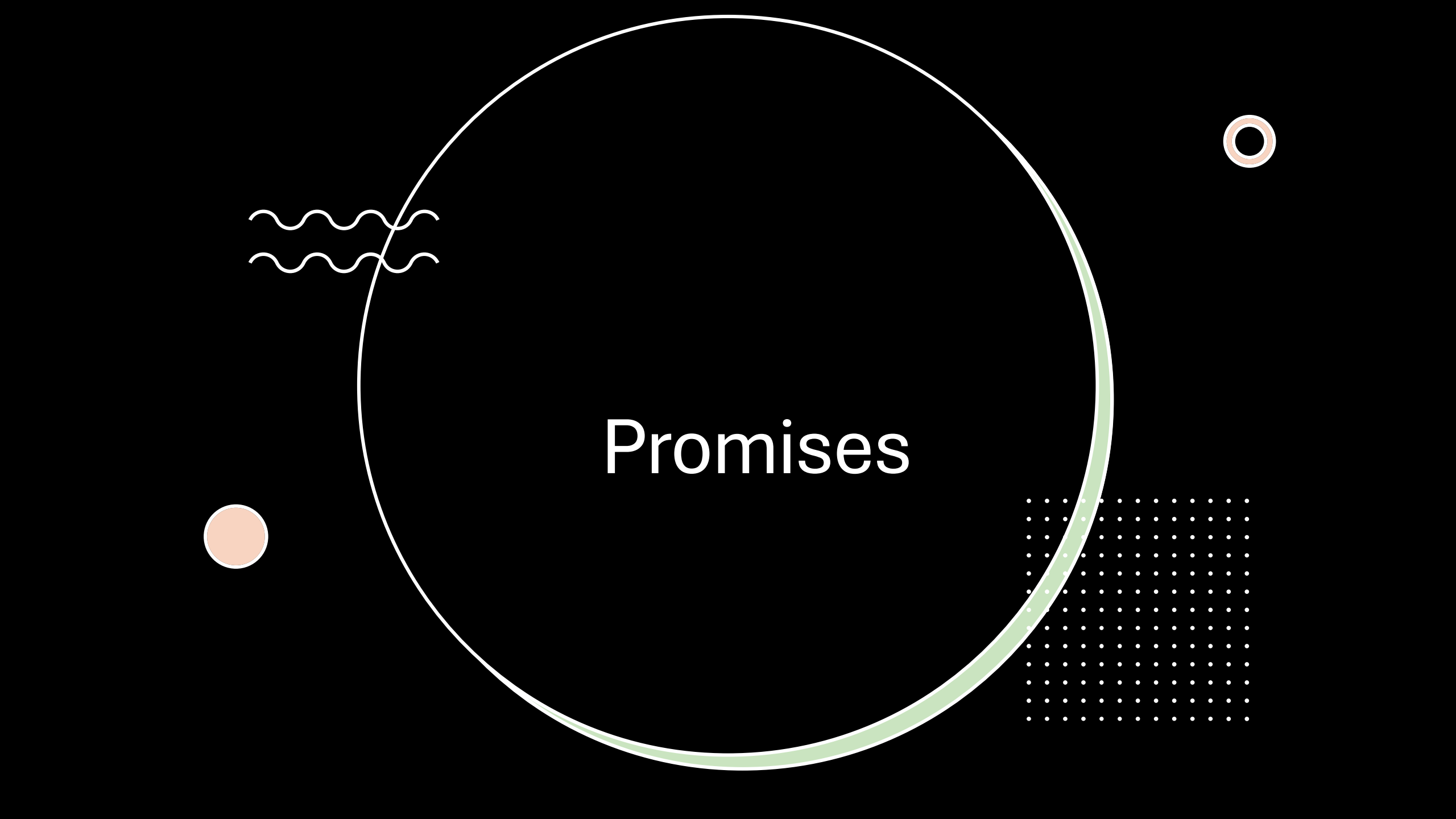
Callback Types

You can incorporate this type into your functions

```
//callbacks
function doHomework(subject:string, callback: () => void) {
    console.log(`Starting my ${subject} homework.`);
    callback();
}
function alertFinished() {
    console.log('Finished my homework');
}

doHomework('math', alertFinished);
```

Parameterized Callbacks




Promises

Promises

Promises are a powerful tool for managing asynchronous operations. The Promise constructor accepts a function with two parameters:

- A function to resolve the promise.
- A function to reject the promise.

```
//Promises are a powerful tool for managing asynchronous operations.
// The Promise constructor accepts a function with two parameters:
// A function to resolve the promise.
// A function to reject the promise. Here's an example:

//create a promise : with a Return Type of string
const doHomework: Promise<string> = new Promise((resolve, reject) => {
    console.log('😀Starting my homework');

    //await 5sec to resolve the promise
    setTimeout(() => {
        reject('💪Finished my homework');
    }, 5000);

    //await 2sec to reject the promise
    setTimeout(() => {
        reject('💀Failed my homework');
    }, 2000);
});
```

Handle Promises: .then() and .catch()

Once a promise is declared, use the .then() and .catch() methods to handle success or failure:

- The .then() method is called when the promise is resolved.
- The .catch() method is called when the promise is rejected

```
//create a promise : with a Return Type of string
const doHomework: Promise<string> = new Promise((resolve, reject) => {
    console.log('😬Starting my homework');

    //await 5sec to resolve the promise
    setTimeout(() => {
        reject('💪Finished my homework');
    }, 5000);

    //await 2sec to reject the promise
    setTimeout(() => {
        reject('💀Failed my homework');
    }, 2000);
});

//handle promises : using .then() and .catch()
doHomework.then((result) => {
    console.log(result);  //outcome of resolve
}).catch((error) => {
    console.log(error);  //outcome of reject
});
```

Handle Promises: Async/Await

TypeScript also supports the async/await syntax, which provides a more readable way to handle promises

- The async keyword indicates that the function returns a promise.
- The await keyword waits for the promise to be resolved.
- The try block handles promise to be resolved.
- The catch block handles rejected promises.

```typescript
//create a promise : with a Return Type of string
const doHomework: Promise<string> = new Promise((resolve, reject) => {
    console.log('😬Starting my homework');

    //await 5sec to resolve the promise
    setTimeout(() => {
        reject('💪Finished my homework');
    }, 5000);

    //await 2sec to reject the promise
    setTimeout(() => {
        reject('💀Failed my homework');
    }, 2000);
});

// handle promises : using async/await
async function doHomeworkAsync() {
    try {
        const result = await doHomework;
        console.log(result); //outcome of resolve
    } catch (error) {
        console.log(error); //outcome of reject
    }
}
```

TypeScript Casting

TypeScript Casting

There are times when working with types when it's necessary to override the variable type, such as when a library provides incorrect types.

Casting is the process of overriding a type

Casting with as

A straightforward way to cast a variable is using the as keyword, which will directly change the type of the given variable.

```
let x: unknown = 'hello';
console.log((x as string).length);
```

```
let x: unknown = 4;
console.log((x as string).length); // prints undefined since numbers don't have a length
```

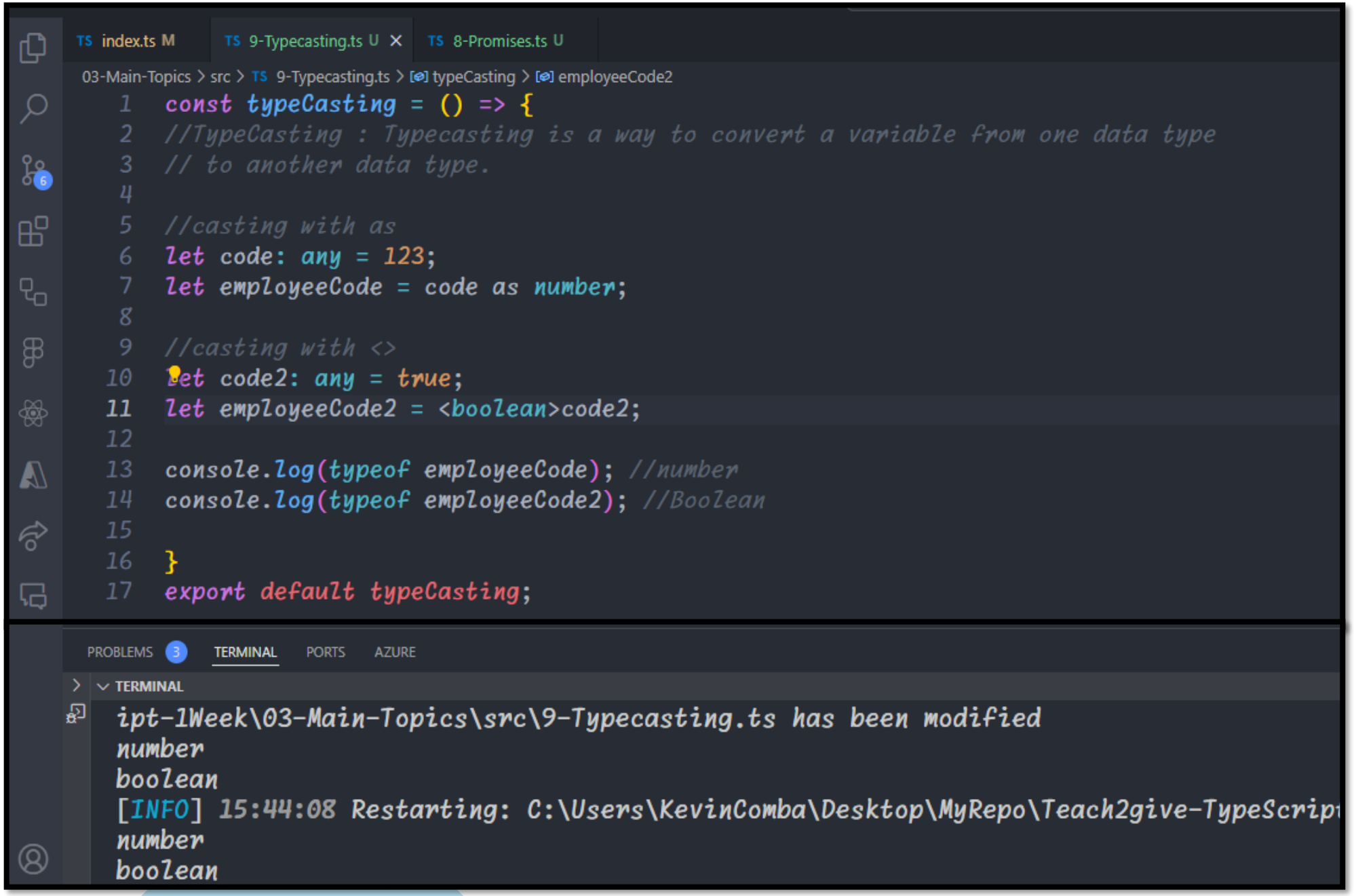
Casting doesn't actually change the type of the data within the variable, for example the following code will not work as expected since the variable x is still holds a number.

Casting with <>

Using <> works the same as casting with as. This type of casting will not work with TSX, such as when working on React files.

```
let x: unknown = 'hello';
console.log((<string>x).length);
```

TypeScript Casting

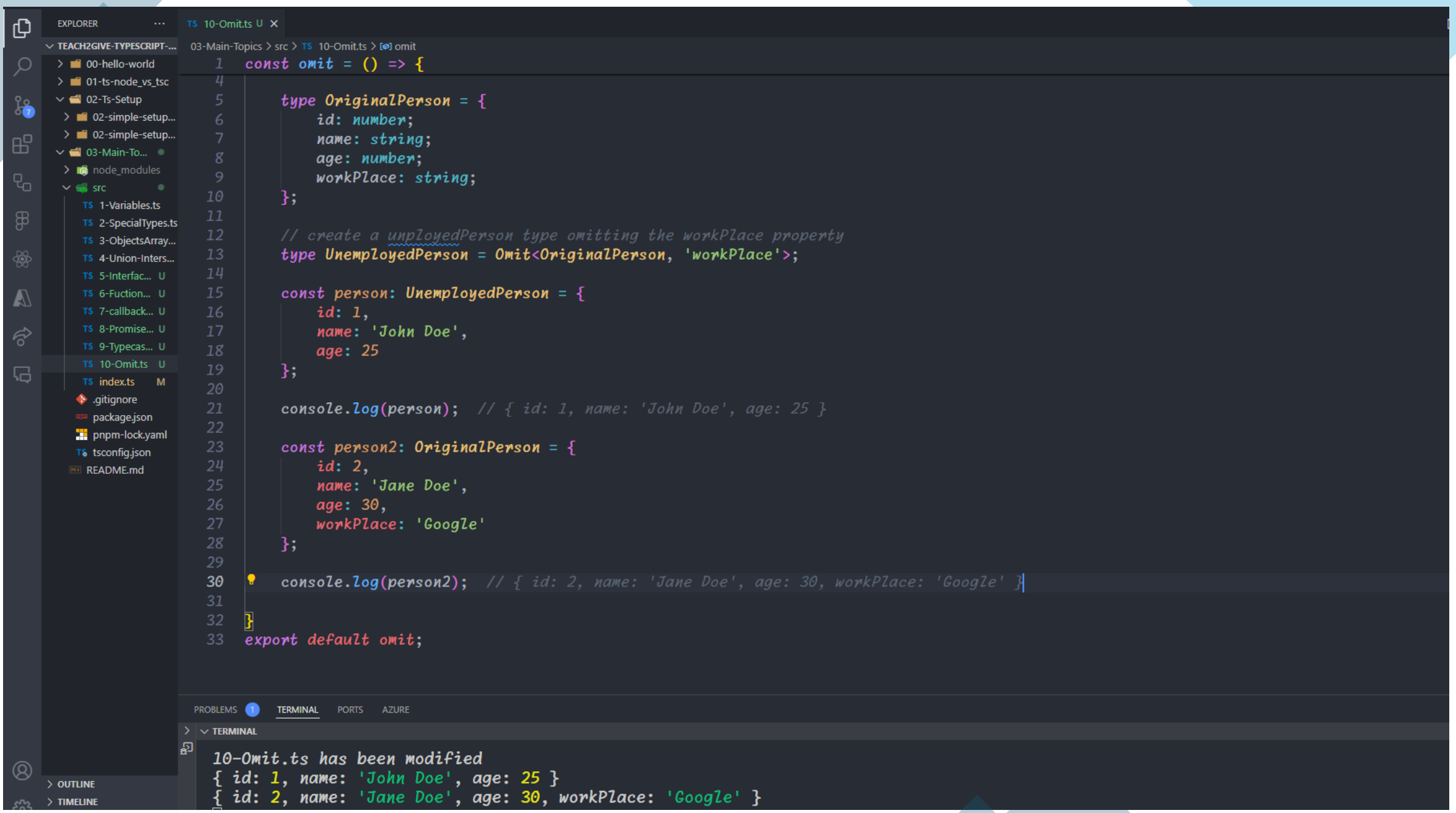


Omit

Omit

- Omit utility type allows you to create a new type by picking all properties from an existing type and then removing specified keys. It’s particularly useful when you want to create a type based on another but excluding certain properties.

```typescript
type OriginalType = {
    id: number;
    name: string;
    age: number;
    // ... other properties
};

// Create a new type by omitting the 'age' property
type NewType = Omit<OriginalType, 'age'>;

// Usage example
const person: NewType = {
    id: 1,
    name: 'Alice',
    // 'age' property is excluded
};
```


EXPLORER
TEACH2GIVE-TYPESCRIPT-...
00-hello-world
01-ts-node_vs_tsc
02-Ts-Setup
02-simple-setup...
02-simple-setup...
03-Main-To...
node_modules
src
1-Variables.ts
2-SpecialTypes.ts
3-ObjectsArray...
4-Union-Inters...
5-Interfac... U
6-Fuction... U
7-callback... U
8-Promise... U
9-Typecas... U
10-Omit.ts U
index.ts M
.gitignore
package.json
pnpm-lock.yaml
tsconfig.json
README.md
OUTLINE
TIMELINE
10-Omit.ts U
03-Main-Topics > src > 10-Omit.ts > omit
const omit = () => {
type OriginalPerson = {
id: number;
name: string;
age: number;
workPlace: string;
};
// create a unployedPerson type omitting the workPlace property
type UnemployedPerson = Omit<OriginalPerson, 'workPlace'>;
const person: UnemployedPerson = {
id: 1,
name: 'John Doe',
age: 25
};
console.log(person); // { id: 1, name: 'John Doe', age: 25 }
const person2: OriginalPerson = {
id: 2,
name: 'Jane Doe',
age: 30,
workPlace: 'Google'
};
console.log(person2); // { id: 2, name: 'Jane Doe', age: 30, workPlace: 'Google' }
}
export default omit;
PROBLEMS 1
TERMINAL
PORTS
AZURE
TERMINAL
10-Omit.ts has been modified
{ id: 1, name: 'John Doe', age: 25 }
{ id: 2, name: 'Jane Doe', age: 30, workPlace: 'Google' }