Typescript

<https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html>

<https://webpack.js.org/concepts/>

<https://blog.logrocket.com/publishing-node-modules-typescript-es-modules/>

<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

Slide:9 Introduction:

TypeScript is Typed JavaScript. TypeScript adds types to JavaScript to help you speed up the development by catching errors before you even run the JavaScript code.

TypeScript [was developed by Microsoft](https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/) to make it [easier to write large code bases](https://www.infoworld.com/article/2614863/microsoft-augments-javascript-for-large-scale-development.html). Essentially, it’s just JavaScript, with static typing.

What’s Static Typing?

If you don’t know, static typing is when the compiler enforces that values use the same type. Here’s an example. This is valid in JavaScript:

let value = 5;

value = "hello";

Here, the type of value changes from a number to a string. In TypeScript, this is forbidden.

let value = 5;

value = "hello"; // error: Type '"hello"' is not assignable to type 'number'.

Slide 10:

Features:

Cross Platform: One of the key features of TypeScript is its cross-platform support. TypeScript code can be compiled to JavaScript, which can run in any environment that supports JavaScript. This includes web browsers, Node.js, and other platforms that can run JavaScript code.

TypeScript's cross-platform support is made possible by its ability to target different ECMAScript (ES) versions. ECMAScript is the standard that defines the syntax and semantics of the JavaScript language. TypeScript can compile code to ES3, ES5, ES6, ES2016, and other versions of ECMAScript, depending on the target environment.

For example, if you are developing a web application that needs to run in older browsers that do not support newer ES features, you can compile your TypeScript code to ES5 or ES3, which will be compatible with those browsers. Alternatively, if you are developing a Node.js application that requires the latest ES features, you can compile your TypeScript code to ES2016 or later.

Object oriented: TypeScript is a strongly typed superset of JavaScript, which means that it adds additional features to JavaScript, including support for object-oriented programming (OOP) concepts. TypeScript supports the core OOP principles, including encapsulation, inheritance, and polymorphism.

Static type checking: Static type checking helps to catch these errors at compile time rather than at runtime, which can improve code quality and reduce bugs.

Optional static Typing: Typescript allows assigning type to a variable. Typescript can also infer the type based on the initialization of the variable. There may be an instance where we want to store a value in a variable but don’t know its type at the time of writing the program. In this case, we want the compiler to opt out of type checking and pass the value without any errors. Typescript has **any** type which allows us to store any type of value and skip type-checking.

let value: any = 5;

console.log(value);

value = "hello";

console.log(value);

DOM Manipulation:

ES6 features: TypeScript has support for many of the new features introduced in ECMAScript 6 (ES6), also known as ECMAScript 2015. In fact, TypeScript has been designed to be a superset of JavaScript, which means that any valid JavaScript code is also valid TypeScript code.

Some of the ES6 features that TypeScript supports include:

1. Arrow functions
2. Classes and inheritance
3. Let and const declarations
4. Template literals
5. Destructuring assignments
6. Spread and rest operators
7. Default and rest parameters
8. Generators and iterators
9. Promises
10. Modules

TypeScript also has support for many other features that have been introduced in subsequent versions of ECMAScript, including ES7, ES8, and ES9. Additionally, TypeScript has its own features that go beyond what is available in JavaScript, such as interfaces, namespaces, decorators, and type annotations.

The ? in that position [marks the property optional](https://www.typescriptlang.org/docs/handbook/functions.html#optional-and-default-parameters).

The ! in that position is the [definite assignment assertion](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-7.html#definite-assignment-assertions). It's sort of a declaration-level version of the [non-null assertion operator](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html#non-null-assertion-operator), but used on a property (can also be used on variables) rather than on an expression.

## **Install TypeScript compiler**

To install the TypeScript compiler, you launch the Terminal on macOS or Linux and Command Prompt on Windows and type the following command:

npm install -g typescript

After the installation, you can type the following command to check the current version of the TypeScript compiler:

tsc --v

It should return the verison like this:

Version 4.0.2

* TypeScript compiler – a Node.js module that compiles TypeScript into JavaScript. If you use JavaScript for node.js, you can install the ts-node module. It is a TypeScript execution and REPL for node.js

npm install -g ts-node

If you installed the ts-node module mentioned in the [setting up TypeScript development environment](https://www.typescripttutorial.net/typescript-tutorial/setup-typescript/), you can use just one command to compile the TypeScript file and execute the output file in one shot:

ts-node app.ts

It’s a community package that enables us to run TypeScript code directly without compiling it first.

to be able to run the ts file using ts-node command you need to first install

npm install @types/node

The @types/node package contains the Node.js type definitions for TypeScript. We need this package to access some Node.js standard libraries.

The “cannot find name ‘console'” error occurs when you try to access the global console object in a TypeScript file. To fix it, install the @types/node NPM package for Node.js environments,

This package is used to load in all type definitions when using typescript in node.

**To compile all TypeScript files in a directory and its subdirectories**, you can use the --project option with the tsc command. This option specifies the path to a directory containing a tsconfig.json file, which is used to configure the TypeScript compiler.

Here's an example command to compile all TypeScript files in a directory and its subdirectories:

tsc --project .

In this command, the --project option specifies the current directory (.), which is assumed to contain a tsconfig.json file. The tsc command will use the tsconfig.json file to configure the compiler, and will compile all TypeScript files in the include paths specified in the tsconfig.json file.tsc --project .

## **What is Type Annotation in TypeScript**

TypeScript uses type annotations to explicitly specify types for identifiers such variables, functions, objects, etc.

TypeScript uses the syntax : type after an identifier as the type annotation, where type can be any valid type.

let name: string = 'John';

let age: number = 25;

let active: boolean = true;

### Arrays

To annotate an [array type](https://www.typescripttutorial.net/typescript-tutorial/typescript-array-type/) you use use a specific type followed by a square bracket : type[] :

let arrayName: type[];

Code language: JavaScript (javascript)

For example, the following declares an array of strings:

let names: string[] = ['John', 'Jane', 'Peter', 'David', 'Mary'];

To specify a type for an object, you use the object type annotation. For example:

let person: {

name: string;

age: number

};

person = {

name: 'John',

age: 25

}; *// valid*

## **Introduction to TypeScript functions**

TypeScript functions are the building blocks of readable, maintainable, and reusable code.

Like JavaScript, you use the function keyword to declare a function in TypeScript:

Let’s see the following add() function example:

function add(a: number, b: number): number {

return a + b;

}

In this example, the add() function accepts two parameters with the [number](https://www.typescripttutorial.net/typescript-tutorial/typescript-number/) type.

When you call the add() function, the TypeScript compiler will check each argument passed to the function to ensure that they are numbers.

In the add() function example, you can only pass numbers into it, not the values of other types.

The following code will result in an error because it passes two strings instead of two numbers into the add() function:

let sum = add('10', '20');

## **Function arguments & return types**

The following shows a function annotation with parameter type annotation and return type annotation:

let greeting : (name: string) => string;

Code language: JavaScript (javascript)

In this example, you can assign any function that accepts a string and returns a string to the greeting variable:

greeting = function (name: string) {

return `Hi ${name}`;

};

Code language: JavaScript (javascript)

The following causes an error because the function that is assigned to the greeting variable doesn’t match with its [function type](https://www.typescripttutorial.net/typescript-tutorial/typescript-function-types/).

greeting = function () {

console.log('Hello');

};

Code language: JavaScript (javascript)

Error:

Type '() => void' is not assignable to type '(name: string) => string'. Type 'void' is not assignable to type 'string'.

Code language: JavaScript (javascript)

## **Introduction to TypeScript any type**

Sometimes, you may need to store a value in a variable. But you don’t know its type at the time of writing the program. And the unknown value may come from a third party API or user input.

In this case, you want to opt-out of the type checking and allow the value to pass through the compile-time check.

# **TypeScript never Type**

The never type is a type that contains no values. Because of this, you cannot assign any value to a variable with a never type.

## **Introduction to TypeScript union type**

Sometimes, you will run into a function that expects a parameter that is either a number or a string. For example:

function add(a: any, b: any) {

if (typeof a === 'number' && typeof b === 'number') {

return a + b;

}

if (typeof a === 'string' && typeof b === 'string') {

return a.concat(b);

}

throw new Error('Parameters must be numbers or strings');

}

Code language: JavaScript (javascript)

In this example, the add() function will calculate the sum of its parameters if they are numbers.

In case the parameters are strings, the add() function will concatenate them into a single string.

If the parameters are neither numbers nor strings, the add() function throws an error.

The problem with the parameters of the add() function is that its parameters have the [any](https://www.typescripttutorial.net/typescript-tutorial/typescript-any-type/) type. It means that you can call the function with arguments that are neither numbers nor strings, the TypeScript will be fine with it.

This code will be compiled successfully but cause an error at runtime:

add(true, false);

Code language: JavaScript (javascript)

To resolve this, you can use the TypeScript union type. The union type allows you to combine multiple types into one type.

For example, the following variable is of type number or string:

let result: number | string;

result = 10; *// OK*

result = 'Hi'; *// also OK*

result = false; *// a boolean value, not OK*

Code language: JavaScript (javascript)

A union type describes a value that can be one of several types, not just two. For example number | string | boolean is the type of a value that can be a number, a string, or a boolean.

Back to the add() function example, you can change the types of the parameters from the any to union like this:

function add(a: number | string, b: number | string) {

if (typeof a === 'number' && typeof b === 'number') {

return a + b;

}

if (typeof a === 'string' && typeof b === 'string') {

return a.concat(b);

}

throw new Error('Parameters must be numbers or strings');

}

**difference between void and undefined as return type in TS**

In TypeScript, void and undefined are two different types of return values that a function can have.

When you declare a function with a return type of void, it means that the function does not return any value. The function may perform some operations or manipulate some data, but it does not return any specific value. For example:

typescript

Copy code

function logMessage(message: string): void {

console.log(message);

}

In the above example, the logMessage function logs a message to the console, but it does not return any value.

On the other hand, when you declare a function with a return type of undefined, it means that the function may return a value of undefined. undefined is a type in TypeScript that represents a variable that has not been assigned a value. For example:

typescript

Copy code

function getUser(id: number): { id: number, name: string } | undefined {

const users = [

{ id: 1, name: 'Alice' },

{ id: 2, name: 'Bob' },

{ id: 3, name: 'Charlie' }

];

const user = users.find(u => u.id === id);

return user;

}

In the above example, the getUser function returns an object that has an id and name property, or it may return undefined if the user with the specified id is not found.

To summarize, the difference between void and undefined as return types in TypeScript is that void indicates that the function does not return any value, while undefined indicates that the function may return a value of undefined.

Difference between JS and TS

1. **There is no concept of data binding in JavaScript**. TypeScript uses types and interfaces to describe data being used.

Optional static typing – hgsf

Dont use optional callback parameters

It is generally not recommended to use optional parameters in callbacks in TypeScript. The reason for this is that optional parameters can cause confusion and errors when calling the callback function.

When a function has optional parameters, it means that those parameters may or may not be present when the function is called. This can be useful in certain situations, but when it comes to callbacks, it can make the code harder to understand and maintain.

For example, consider the following code that uses a callback function with an optional parameter:

typescript

Copy code

function fetchData(callback: (data?: any) => void) {

// fetch data from server

const data = { /\* some data \*/ };

callback(data);

}

Here, the fetchData function takes a callback function that may or may not have a parameter of any type. This can make it difficult to know whether or not to expect a parameter when calling the callback.

Instead of using optional parameters, it is recommended to use a union type to define the possible parameters that the callback function can take. For example:

typescript

Copy code

type CallbackFunction = (data: any) => void | (() => void);

function fetchData(callback: CallbackFunction) {

// fetch data from server

const data = { /\* some data \*/ };

callback(data);

}

Here, the CallbackFunction type specifies that the callback function can take a parameter of any type or no parameter at all.

Using a union type like this can make the code more explicit and easier to understand when calling the callback function. It can also help to prevent errors and bugs caused by optional parameters.

TS function parameters

TS Decorators

TypeScript decorators can be thought of as special functions that you can attach to classes, methods, properties, or parameters to add extra functionality or metadata to them. Imagine decorators as labels or tags you can put on different parts of your code.

1. **Class Decorators**: Think of them as stickers you put on a book cover. These stickers can change the entire book's appearance or behavior. For example, you might have a "Fantasy" sticker on a book, and that sticker tells readers that this book is a fantasy genre.
2. **Method Decorators**: Consider them as notes you attach to specific pages in the book. These notes can give readers additional information about that particular page. For instance, you can put a "Caution" note on a page to warn readers of something important.
3. **Property Decorators**: Think of them as labels you attach to specific paragraphs or sentences inside the book. These labels can highlight certain sentences or provide extra context. For example, you can label a particular sentence as "Important."
4. **Parameter Decorators**: Imagine them as marks you make on certain words or phrases in the book's text. These marks can help readers understand how to use those words or phrases. For instance, you can underline a word to show it's a keyword.
5. **Metadata**: Think of metadata as bookmarks in your book. You can add bookmarks to remember specific pages or sections. In programming, decorators can store additional information (metadata) that can be used later to influence how your code behaves.

In essence, TypeScript decorators allow you to modify or enhance your code by attaching labels, notes, or bookmarks to various parts of your program. These labels and notes can affect how your code functions or provide additional information for developers to understand and work with the code more effectively.

Adding Metadata to decorator

npm install reflect-metadata

To use metadata-related functions like **Reflect.getMetadata**, you need to import the **"reflect-metadata"** package and enable the **experimentalDecorators** and **emitDecoratorMetadata** options in your TypeScript configuration.

Here's how to fix the issue:

1. Install the **"reflect-metadata"** package:

bashCopy code

npm install reflect-metadata

1. Import **"reflect-metadata"** at the top of your TypeScript file:

typescriptCopy code

import 'reflect-metadata';

1. Ensure that your TypeScript configuration (**tsconfig.json**) has the following settings:

jsonCopy code

{ "compilerOptions": { "experimentalDecorators": true, "emitDecoratorMetadata": true // ... } }

## Grunt Task Runner

What is Grunt Task Runner?

We’ll start by installing the Grunt command line interface (CLI).

|  |  |
| --- | --- |
| 1 | > npm install grunt-cli -g |

It does not install the task runner itself. This is only a simple portion that provides command line access to the Grunt task runner, which will be installed locally alongside your project. That way, multiple projects can use their own version of Grunt.

To install the Grunt task runner, let’s run the following command:

|  |  |
| --- | --- |
| 1 | > npm install grunt --save-dev |

The last bit of installation is to create “gruntfile.js” in the project root folder. This is the node.js script that will contain definitions and configuration of all tasks that we want to run for this project.

Tasks are what Grunt executes and runs. A task can represent a particular thing to do, like a step of your build process or a sequence of steps. Whenever you run Grunt, you specify one or more tasks to run.

To run a particular task, you can use following command: > grunt [task\_name],

where [task\_name] is the name of a task to run. Grunt will attempt to run a task named “default” if the task name is not specified.

So, **> grunt default**  and **> grunt**    will do the same thing.

### Gruntfile

Tasks are specified and configured in the gruntfile. It has a simple structure:

### Plugins

The list of plugins is available at [gruntjs.com](https://gruntjs.com/plugins). Plugins are installed using NPM. Once a plugin is installed, you can call  grunt.loadNpmTasks() method, passing a plugin name as a first parameter, to load tasks from the plugin:

How to compile Typescript with Grunt:

To compile TypeScript with Grunt, you can follow these steps:

Install the required packages:

css

Copy code

npm install grunt grunt-ts --save-dev

Create a Gruntfile.js in the root directory of your project and configure it:

js

Copy code

module.exports = function(grunt) {

grunt.initConfig({

ts: {

app: {

files: [{

src: ['src/\*\*/\*.ts'], // path to your TypeScript files

dest: 'dist/' // output directory for compiled JavaScript files

}],

options: {

module: 'commonjs', // use commonjs module system

target: 'es5', // compile to ECMAScript 5

sourceMap: true // generate source maps

}

}

}

});

grunt.loadNpmTasks('grunt-ts');

grunt.registerTask('default', ['ts']);

};

Run Grunt by typing grunt in your terminal. This will compile your TypeScript files and output them to the specified destination folder.

You can also add additional configuration options to the ts task as needed. For example, you can configure Grunt to watch for changes in your TypeScript files and automatically recompile them by adding the following to the ts task:

js

Copy code

watch: 'src/\*\*/\*.ts'

And then running grunt watch in your terminal.

Npx tsc –init: generate tsconfig.json

npm install grunt grunt-ts --save-dev

npm install -g grunt-cli

npm install grunt-typescript --save-dev

npm install grunt-contrib-watch --save-dev

Webpack:

To compile TypeScript using Webpack, you can follow these steps:

Install the required dependencies:

npm install --save-dev typescript ts-loader webpack webpack-cli

Create a tsconfig.json file at the root of your project to configure TypeScript:

{

"compilerOptions": {

"target": "es5",

"module": "es6",

"strict": true,

"esModuleInterop": true,

"sourceMap": true

},

"include": [

"src/\*\*/\*.ts"

]

}

Create a webpack.config.js file at the root of your project to configure Webpack:

const path = require('path');

module.exports = {

entry: './src/index.ts',

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'bundle.js'

},

resolve: {

extensions: ['.ts', '.js']

},

module: {

rules: [

{

test: /\.ts$/,

use: 'ts-loader',

exclude: /node\_modules/

}

]

}

};

Create a src/index.ts file to write TypeScript code:

class Greeter {

private name: string;

constructor(name: string) {

this.name = name;

}

greet() {

console.log(`Hello, ${this.name}!`);

}

}

const greeter = new Greeter('World');

greeter.greet();

Run webpack in the terminal to compile the TypeScript code:

npx webpack --config webpack.config.js

This will generate a dist/bundle.js file that you can use in your application.

Gulp:

npm install -g gulp

To compile TypeScript with Gulp, you can follow these steps:

Install the required packages:

css

Copy code

npm install gulp gulp-typescript --save-dev

Create a gulpfile.js in the root directory of your project and configure it:

js

Copy code

const gulp = require('gulp');

const ts = require('gulp-typescript');

const tsProject = ts.createProject('tsconfig.json');

function compileTypeScript() {

const tsResult = tsProject.src().pipe(tsProject());

return tsResult.js.pipe(gulp.dest('dist/'));

}

function watchTypeScript() {

gulp.watch('src/\*\*/\*.ts', compileTypeScript);

}

exports.default = gulp.series(compileTypeScript, watchTypeScript);

Run Gulp by typing gulp in your terminal. This will compile your TypeScript files and output them to the specified destination folder.

Keep the Gulp task running and it will watch for changes in your TypeScript files and automatically recompile them.

You can also run the watch task separately by typing gulp watch in your terminal. This will only watch for changes in your TypeScript files and not compile them immediately.

You can also add additional configuration options to the ts task as needed. For example, you can configure Gulp to exclude certain files or folders from the watch task by adding the following to the ts task:

js

Copy code

function compileTypeScript() {

const tsResult = tsProject.src().pipe(tsProject());

return tsResult.js.pipe(gulp.dest('dist/'));

}

function watchTypeScript() {

gulp.watch(['src/\*\*/\*.ts', '!src/\*\*/node\_modules/\*\*'], compileTypeScript);

}

exports.default = gulp.series(compileTypeScript, watchTypeScript);

This will exclude all files and folders under the node\_modules folder from the watch task.

Browserify:

Browserify is an open-source JavaScript bundler tool that allows developers to write and use Node.js-style modules that compile for use in the browser.

## What is Browserify?

Browserify is a build tool that allows you to use Node.js modules directly in the browser. You can use require with Browserify the same way you can with Node. Browserify provides a common way to structure all of your [JavaScript](https://www.stackchief.com/tutorials/JavaScript%20ES6%20Intro) code by bundling dependencies into a single file that can be referenced within a <script> tag in the browser.

Now let’s move this project from Node to the browser. To do this, we’d like to bundle all our modules into one JavaScript file. Fortunately, that’s exactly what Browserify does. Even better, it lets us use the CommonJS module system used by Node, which is the default TypeScript emit. That means our TypeScript and Node setup will transfer to the browser basically unchanged.

First, install browserify, [tsify](https://www.npmjs.com/package/tsify), and vinyl-source-stream. tsify is a Browserify plugin that, like gulp-typescript, gives access to the TypeScript compiler. vinyl-source-stream lets us adapt the file output of Browserify back into a format that gulp understands called [vinyl](https://github.com/gulpjs/vinyl).

Typescript with JSX

To compile a .tsx file to JavaScript and use it in an HTML file, you can use a build tool like webpack or rollup. Here's an example using webpack:

Install required dependencies: You will need to install the following dependencies as dev dependencies using npm or yarn:

npm install webpack webpack-cli ts-loader typescript react react-dom @types/react @types/react-dom --save-dev

Create a webpack configuration file: Create a webpack.config.js file in the root directory of your project with the following configuration:

javascript

Copy code

const path = require('path');

module.exports = {

entry: './src/index.tsx',

mode: 'production',

module: {

rules: [

{

test: /\.tsx?$/,

use: 'ts-loader',

exclude: /node\_modules/,

},

],

},

resolve: {

extensions: ['.tsx', '.ts', '.js'],

},

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist'),

},

};

Here, we define the entry point for our application as ./src/index.tsx, set the mode to 'production', and configure the ts-loader to compile TypeScript files. We also configure the output path to be dist/bundle.js.

Create a .tsx file: Create a file with a .tsx extension and write your TypeScript code that contains JSX syntax. For example:

tsx

Copy code

import React from 'react';

import ReactDOM from 'react-dom';

const App = () => {

return <h1>Hello, world!</h1>;

};

ReactDOM.render(<App />, document.getElementById('root'));

Compile the TypeScript code: Run the following command in your terminal to compile the TypeScript code and generate the bundle.js file:

npx webpack --config webpack.config.js

This will generate the bundle.js file in the dist directory.

Create an HTML file: Create an HTML file in the root directory of your project, and add the following code:

html

Copy code

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>My App</title>

</head>

<body>

<div id="root"></div>

<script src="./dist/bundle.js"></script>

</body>

</html>

This code creates a div element with an id of root where the React app will be rendered. The script tag at the end loads the bundle.js file generated by webpack.

Open the HTML file in a browser: Open the HTML file in a browser, and you should see the "Hello, world!" message rendered on the page.

Note that this is just a basic example of how to compile and use a .tsx file in an HTML file using webpack. You can customize the configuration and file paths to suit your specific needs.

Resolve property in webpack.config.js

In Webpack, the resolve property in the configuration file is used to specify how Webpack resolves modules. One of the options for the resolve property is the extensions property, which is an array of file extensions that Webpack will try when resolving module imports.

Here's an example of how to use the extensions property in a webpack.config.js file:

javascript

Copy code

module.exports = {

// ...

resolve: {

extensions: ['.js', '.jsx', '.ts', '.tsx']

}

};

In this example, the resolve property is set to an object with an extensions property that is an array of file extensions. The file extensions .js, .jsx, .ts, and .tsx are included in the array, which means that Webpack will try these extensions in order when resolving module imports.

This can be useful for importing modules without specifying their file extension every time, which can make the code cleaner and easier to read. For example, if you have a file named MyComponent.jsx, you can import it in another file like this:

javascript

Copy code

import MyComponent from './MyComponent';

Without the extensions property set, you would need to specify the file extension in the import statement like this:

javascript

Copy code

import

Angular

app.module.ts is a TypeScript file that defines the root module of an Angular application. The module is responsible for bootstrapping the application, defining its components, services, directives, pipes, and other application features, and wiring them together.

The app.module.ts file typically starts with importing the required Angular modules and other third-party modules, followed by the definition of the @NgModule decorator, which is used to define the metadata of the module. The @NgModule decorator takes an object as its argument, which includes the following properties:

declarations: An array of components, directives, and pipes that are declared in this module.

imports: An array of modules that this module depends on.

providers: An array of services that are provided by this module.

bootstrap: An array of components that are used to bootstrap the application.

exports: An array of components, directives, and pipes that are exported from this module and can be used in other modules.

**BrowserModule**

For apps that run in the browser, import **BrowserModule** in the root AppModule because it **provides services that are essential to launch and run a browser app."** and "BrowserModule imports CommonModule, which contributes many common directives such as ngIf and ngFor.