

Cheat Sheet - TypeScript

TypeScript Cheat Sheet				
Setup Install TS globally on your machine <pre>\$ npm i -g typescript</pre> Check version <pre>\$ tsc -v</pre> Create the tsconfig.json file <pre>\$ tsc --init</pre> Set the root (to compile TS files from) and output (for the compiled JS files) directories in tsconfig.json <pre>"rootDir": "./src", "outDir": "./public",</pre>	Primitive Types There are 7 primitive types in JS: string, number, bigint, boolean, undefined, null, symbol. Explicit type annotation <pre>let firstName: string = 'Danny'</pre> If we assign a value (as above), we don't need to state the type - TS will infer it ("implicit type annotation") <pre>let firstName = 'Danny'</pre>	Arrays We can define what kind of data an array can contain <pre>let ids: number[] = []; ids.push(1); ids.push("2"); // Error</pre> Use a union type for arrays with multiple types <pre>let options: (string number)[]; options = [10, 'UP'];</pre> If a value is assigned, TS will infer the types in the array. <pre>let person = ['Delia', 48]; person[0] = true; // Error - only strings or numbers allowed</pre>	Interfaces Interfaces are used to describe objects. Interfaces can always be reopened & extended, unlike Type Aliases. Notice that 'name' is 'readonly' <pre>interface Person { name: string; isProgrammer: boolean; }</pre> <pre>let p1: Person = { name: 'Delia', isProgrammer: false, }; p1.name = 'Del'; // Error - read only</pre> Two ways to describe a function in an interface <pre>interface Speech { sayHi(name: string): string; sayBye: (name: string) => string; }</pre> <pre>let speech: Speech = { sayHi: function (name: string) { return 'Hi ' + name; }, sayBye: (name: string) => 'Bye ' + name, };</pre> Extending an interface <pre>interface Animal { name: string; }</pre> <pre>interface Dog extends Animal { breed: string; }</pre>	Generics Generics allow for type safety in components where the arguments & return types are unknown ahead of time. <pre>interface HasLength { length: number; }</pre> <pre>// logLength accepts all types with a length property const logLength = <T extends HasLength> (a: T) => { console.log(a.length); };</pre> <pre>// TS "captures" the type implicitly logLength('Hello'); // 5</pre> <pre>// Can also explicitly pass the type to T logLength<number>([1, 2, 3]); // 3</pre> Declare a type, T, which can change in your interface. <pre>interface Dog<T> { breed: string; treats: T; }</pre> <pre>// We have to pass in a type argument let labrador: Dog<string> = { breed: 'labrador', treats: 'chew sticks, tripe', };</pre> <pre>let scottieDog: Dog<string[]> = { breed: 'scottish terrier', treats: ['turkey', 'haggis'], };</pre>
Compiling Compile a specified TS file into a JS file of the same name, into the same directory (i.e. index.ts to index.js). <pre>\$ tsc index.ts</pre> Tell tsc to compile specified file whenever a change is saved by adding the watch flag (-w) <pre>\$ tsc index.ts -w</pre> Compile specified file into specified output file <pre>\$ tsc index.ts --outfile out/script.js</pre> If no file is specified, tsc will compile all TS files in the "rootDir" and output in the "outDir". Add -w to watch for changes. <pre>\$ tsc -w</pre>	Union Types A variable that can be assigned more than one type <pre>let age: number string; age = 26; age = "26";</pre>	Tuples A tuple is a special type of array with fixed size & known data types at each index. They're stricter than regular arrays. <pre>let options: [string, number]; options = ['UP', 10];</pre>	Enums A set of related values, as a set of descriptive constants <pre>enum ResourceType { BOOK, FILE, FILM, }</pre> <pre>ResourceType.BOOK; // 0 ResourceType.FILE; // 1</pre>	
Dynamic Types The any type basically reverts TS back to JS. <pre>let age: any = 100; age = true;</pre>	Literal Types We can refer to specific strings & numbers in type positions <pre>let direction: 'UP' 'DOWN'; direction = 'UP';</pre>	Functions We can define the types of the arguments, and the return type. Below, :string could be omitted because TS would infer the return type. <pre>function circle(diam: number): string { return 'Circumf = ' + Math.PI * diam; }</pre> The same function as an ES6 arrow <pre>const circle = (diam: number): string => 'Circumf = ' + Math.PI * diam;</pre> If we want to declare a function, but not define it, use a function signature <pre>let sayHi: (name: string) => void; sayHi = (name: string) => console.log('Hi ' + name); sayHi('Danny'); // Hi Danny</pre>	The DOM & Type Casting TS doesn't have access to the DOM, so use the non-null operator, !, to tell TS the expression isn't null or undefined <pre>const link = document.querySelector('a');</pre> If an element is selected by id or class, we need to tell TS what type of element it is via Type Casting <pre>const form = document.getElementById('signup-form') as HTMLFormElement;</pre>	
Literal Types We can refer to specific strings & numbers in type positions <pre>let direction: 'UP' 'DOWN'; direction = 'UP';</pre>	Objects Objects in TS must have all the correct properties & value types <pre>let person: { name: string; isProgrammer: boolean; }; person = { name: 'Danny', isProgrammer: true, }; person.age = 26; // Error - no age prop on person object person.isProgrammer = 'yes'; // Error - should be boolean</pre>	Type Aliases Allow you to create a new name for an existing type. They can help to reduce code duplication. They're similar to interfaces, but can also describe primitive types. <pre>type StringOrNum = string number; let id: StringOrNum = 24;</pre>	Narrowing Occurs when a variable moves from a less precise type to a more precise type <pre>let age = getUserAge(); age // string number if (typeof age === 'string') { age; // string }</pre>	
Strict Mode In tsconfig.json, it is recommended to set strict to true. One helpful feature of strict mode is No Implicit Any: <pre>// Error: Parameter 'a' implicitly has an 'any' type. function logName(a) { console.log(a.name); }</pre> <i>By @DoableDanny</i>				

Revision #1

Created 10 November 2022 15:44:15 by gpatruno

Updated 10 November 2022 15:46:49 by gpatruno