TypeScript

by Fred Bolder

Foreword

TypeScript is a programming language that adds static typing to JavaScript. It is a superset of JavaScript. A TypeScript file must have the extension .ts or .tsx (extended). TypeScript can not run in a browser or Node.js, but TypeScript files can be converted into JavaScript files by using the TypeScript Compiler (tsc).

This e-book contains a lot of handy examples.

Installing

Globally
```
npm install -g typescript
tsc -v
tsc --init
```

Locally (project)
```
npm install typescript --save-dev
npx tsc -v
npx tsc --init
```

Existing React project
```
npm install --save typescript @types/node @types/react @types/react-dom @types/jest
npx tsc -v
npx tsc --init

In tsconfig.json change
// "jsx": "preserve",
to
"jsx": "react",
```

Adding additional types
```
npm i @types/jsonwebtoken
```
https://www.npmjs.com/search?q=%40types

Tailwind
```
npx tailwindcss -i ./src/index.css -o ./dist/output.css --watch
```

Compiling

tsc <file>

When there is no file specified, all files in the current folder and subfolders will be compiled (default options in tsconfig.json).
If TypeScript is installed locally, the tsc command must be preceded by the npx command.

npx tsc <file>

Example:

Original TypeScript file myFile.ts:

```
let msg: string = "Hello";
console.log(msg);
```

Compile the TypeScript file:

```
npx tsc myFile.ts
```

JavaScript file myFile.js created by tsc (can be different depending on tsconfig.json):

```
var msg = "Hello";
console.log(msg);
```

Run the JavaScript file:

```
node myFile
```

Compiling a single module file

npx tsc --module <target> <file>

target = None, CommonJS, AMD, UMD, System, ES6, ES2015 or ESNext

https://www.tsmean.com/articles/learn-typescript/typescript-module-compiler-option/

Types

Example 1:

```
let firstName: string = "John";
let age: number = 25;
let married: boolean = true;
console.log(`Name: ${firstName}, Age: ${age}, Married: ${married}`);
// Output: Name: John, Age: 25, Married: true
```

Example 2:

```
let id: number | string;
id = 253;
console.log("ID: ", id); // Output: ID: 253
id = "zed7t";
console.log("ID: ", id); // Output: ID: zed7t
```

Example 3:

```
type Id = number | string;
let myId: Id = 25;
console.log("ID: ", myId); // Output: ID: 25
myId = "gKef2";
console.log("ID: ", myId); // Output: ID: gKef2
```

Example 4:

```
type TypeString = string;
let str1: string = "";
let str2: TypeString = "";
let num: number = 123;
console.log(typeof str1); // Output: string
console.log(typeof str2); // Output: string
str2 = num.toString();
str1 = str2;
console.log(str1); // Output: 123
```

Example 5:

```
type User = {
    name: string,
    age: number
};
let myUser: User = { name: "John", age: 25};
console.log(`Name: ${myUser.name}, Age:  ${myUser.age}`);
// Output: Name: John, Age:  25
console.log(typeof myUser); // Output: object
```

Example 6:

```
type TPoint = {
    x: number,
    y: number
}

type TProps = {
    id: number,
    color: string
}

type TLine = TProps & {
    ptStart: TPoint,
    ptEnd: TPoint
}

type TCircle = TProps & {
    ptCenter: TPoint,
    radius: number
}

let myCircle: TCircle = {
    id: 1,
    color: "blue",
    ptCenter: { x: 100, y: 50 },
    radius: 10
};

let myLine: TLine = {
    color: "white",
    ptStart: { x: 0, y: 0 },
    ptEnd: { x: 100, y: 50 },
    id: 2,
};

console.log(myCircle);
// Output: { id: 1, color: 'blue', ptCenter: { x: 100, y: 50 }, radius: 10 }
console.log(myLine);
// Output:
// {
//   color: 'white',
//   ptStart: { x: 0, y: 0 },
//   ptEnd: { x: 100, y: 50 },
//   id: 2
// }
```

Example 7:

```
console.log(typeof "TEST"); // Output: string
console.log(typeof("TEST")); // Output: string
```

Example 8:

```
let a: any = 123;  // Try to avoid using the type any
console.log(a); // Output: 123
a = "Test";
console.log(a); // Output: Test
```

Example 9:

```
type Color = "Red" | "Green";

function printColor(color: Color) {
    console.log(`Color: ${color}`); // Output: Color: Red
}

printColor("Red");
```

Type inference

There is no need to annotate everything.

Example 1:

```
let firstName: string = "Fred";
console.log(firstName);
```

TypeScript understands that "Fred" is a string.

```
let firstName = "Fred";
console.log(firstName);
```

Partial

Partial tells TypeScript that every property in the type is optional.

Example 1:

```
type Person = {
    firstName: string,
    lastName: string,
    age: number,
}

let person1: Person = {
    firstName: "John",
    lastName: "Smith",
    age: 25,
}

let person2: Partial<Person> = {
    firstName: "Mary",
}

console.log(person1);
// Output: { firstName: 'John', lastName: 'Smith', age: 25 }
console.log(person2); // Output: { firstName: 'Mary' }
```

Required

Required tells TypeScript that every property in the type is required.

Example 1:

```
type Person = {
    firstName: string,
    lastName?: string,
    age?: number,
}

let person1: Person = {
    firstName: "Mary",
}

let person2: Required<Person> = {
    firstName: "John",
    lastName: "Smith",
    age: 25,
}

console.log(person1); // Output: { firstName: 'Mary' }
console.log(person2);
// Output: { firstName: 'John', lastName: 'Smith', age: 25 }
```

Omit

Example 1:

```
enum gender {
    M,
    F,
}

type Person = {
    firstName: string,
    lastName: string,
    age: number,
    gender: gender,
}

type PersonNameOnly = Omit<Person, "age" | "gender">;

let person1: PersonNameOnly = {
    firstName: "Kim",
    lastName: "Wilde",
}

let person2: Person = {
    firstName: "John",
    lastName: "Smith",
    age: 25,
    gender: gender.M,
}

let person3: Omit<Person, "gender"> = {
    firstName: "Mary",
    lastName: "Jones",
    age: 30,
}

console.log(person1); // Output: { firstName: 'Kim', lastName: 'Wilde' }
console.log(person2);
// Output: { firstName: 'John', lastName: 'Smith', age: 25, gender: 0 }
console.log(person3);
// Output: { firstName: 'Mary', lastName: 'Jones', age: 30 }
```

Pick

Example 1:

```
enum gender {
    M,
    F,
}

type Person = {
    firstName: string,
    lastName: string,
    age: number,
    gender: gender,
}

let person1: Pick<Person, "firstName" | "lastName"> = {
    firstName: "Kim",
    lastName: "Wilde",
}

console.log(person1); // Output: { firstName: 'Kim', lastName: 'Wilde' }
```

NonNullable

Example 1:

```
type Person = {
    name: string,
    age?: number,
} | null;

function printPerson(person: NonNullable<Person>) {
  if (person.name) {
    console.log(person.name);
  }
  if (person.age) {
    console.log(person.age.toString());
  }
}

printPerson({ name: "Fred" }); // Output: Fred
// The following lines (if uncommented) would give an error
//printPerson(undefined);
//printPerson(null);
```

Arrays

Example 1:

```
let nums: number[] = [1, 2, 3.14];
console.log(nums); // Output: [ 1, 2, 3.14 ]
```

Example 2:

```
let nums: Array<number> = [1, 2, 3.14];
console.log(nums); // Output: [ 1, 2, 3.14 ]
```

Example 3:

```
let arr: (boolean | number)[] = [25, false, true, 3, 2.5];
console.log(arr); // Output: [ 25, false, true, 3, 2.5 ]
```

Example 4:

```
let arr: Array<string | number> = ["Test", 123, "ABC"];
console.log(arr); // Output: [ 'Test', 123, 'ABC' ]
```

Example 5:

```
let arr: number[][] = [[1, 2], [3, 4, 5]];
console.log(arr); // Output: [ [ 1, 2 ], [ 3, 4, 5 ] ]
console.log(arr[0][0]); // Output: 1
console.log(arr[0][1]); // Output: 2
console.log(arr[0][2]); // Output: undefined
console.log(arr[1][0]); // Output: 3
console.log(arr[1][1]); // Output: 4
console.log(arr[1][2]); // Output: 5
```

Example 6:

```
let arr: number[] = [1, 2, 3];
arr.length =  5;
arr.push(4);
console.log(arr); // Output: [ 1, 2, 3, <2 empty items>, 4 ]
```

Example 7:

```
const arr1: number[] = [1, 2, 3];

function readOnly<T>(arr: T[]): ReadonlyArray<T> {
    const readOnlyArr: ReadonlyArray<T> = arr;
    return readOnlyArr;
}

console.log(arr1); // Output: [ 1, 2, 3 ]
arr1[0] = 4;
console.log(arr1); // Output: [ 4, 2, 3 ]
const arr2 = readOnly(arr1);
console.log(arr2); // Output: [ 4, 2, 3 ]
//arr2[0] = 5; // would give an error
arr1[0] = 5;
console.log(arr1); // Output: [ 5, 2, 3 ]
console.log(arr2); // Output: [ 5, 2, 3 ]
```

Example 8:

```
const arr: number[] = [1, 2, 3];

function doSomething(arr: readonly number[]): void {
    console.log(arr[0]);
    //arr[0] = 4; // would give an error
}

doSomething(arr); // Output: 1
arr[0] = 4;
doSomething(arr); // Output: 4
```

Example 9:

```
type Person = {
    name: string,
    hobbies?: string[],
}

const person1: Person = {
    name: "Fred",
    hobbies: ["programming", "singing"],
}

const person2: Person = {
    name: "John",
}

const person3: Person = {
    name: "Kim",
    hobbies: [],
}

function printFirstHobby(person: Person) {
    console.log(person.hobbies ? person.hobbies[0] : undefined);
}

printFirstHobby(person1); // Output: programming
printFirstHobby(person2); // Output: undefined
printFirstHobby(person3); // Output: undefined
```

Tuples

Example 1:

```
let user: [number, string] = [1, "Fred"];
console.log(user); // Output: [ 1, 'Fred' ]
console.log(user[0]); // Output: 1
console.log(user[1]); // Output: Fred
```

Example 2:

```
let users: [number, string][] = [[1, 'Fred']];
console.log(users); // Output: [ [ 1, 'Fred' ] ]
users.push([2, 'John']);
console.log(users); // Output: [ [ 1, 'Fred' ], [ 2, 'John' ] ]
console.log(users[users.length - 1]); // Output: [ 2, 'John' ]
```

Example 3:

```
let user: [id: number, name: string] = [243, "Linda"];
console.log(user); // Output: [ 243, 'Linda' ]
console.log(user[0]); // Output: 243
console.log(user[1]); // Output: Linda
```

key-value

Example 1:

```
const numbers = {
    one: 1,
    two: 2,
    three: 3,
}

function numberStringToNumber(key: string) {
    if (numbers.hasOwnProperty(key as keyof typeof numbers)) {
        return numbers[key as keyof typeof numbers]
    } else {
        return null;
    }
}

console.log(numberStringToNumber("two")); // Output: 2
const three: string = "three";
console.log(numberStringToNumber("three")); // Output: 3
console.log(numberStringToNumber("ten")); // Output: null
```

Functions

Example 1:

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

console.log(add(1, 2)); // Output: 3
```

Example 2:

```
function showMessage(msg: string): void {
    console.log(msg);
}

showMessage("Hello!"); // Output: Hello!
```

Example 3:

```
let add = (a: number, b: number, c?: number): number => {
    let result: number;

    result = a + b;
    if (c) {
        result += c;
    }
    return result;
}

console.log(add(25, 50)); // Output: 75
console.log(add(25, 50, 10)); // Output: 85
```

Example 4:

```
let add = (a: number, b: number, c: number = 0): number => {
    let result: number;

    result = a + b + c;
    return result;
}

console.log(add(25, 50)); // Output: 75
console.log(add(25, 50, 10)); // Output: 85
```

Example 5:

```
function add(num: number, ...nums: number[]): number {
    let result: number = num;

    for (let i = 0; i < nums.length; i++) {
        result += nums[i];
    }
    return result;
}

console.log(add(25, 50)); // Output: 75
console.log(add(25, 50, 10)); // Output: 85
console.log(add(1, 2, 3, 4)); // Output: 10
```

Generics

Example 1:

```
function getLast<T>(arr: T[]): T {
    return arr[arr.length - 1];
}

let strings: string[] = ["John", "Fred"];
console.log(getLast<string>(strings)); // Output: Fred
let numbers: number[] = [1, 2, 3];
console.log(getLast<number>(numbers)); // Output: 3
let empty: number[] = [];
console.log(getLast(empty)); // undefined
```

Interfaces

Example 1:

```
interface IPerson {
    firstName: string,
    lastName: string,
    fullName(): string
}

const person: IPerson = {
    firstName: "John",
    lastName: "Smith",
    // You can not use an arrow function in combination with 'this'.
    fullName(): string { return `${this.firstName} ${this.lastName}` }
}

console.log(person.fullName()); // Output: John Smith
```

Example 2:

```
interface FunctionType {
    (msg: string): void
}

function showMessage(msg: string): void {
    console.log(msg);
}

function showMessageUppercase(msg: string): void {
    console.log(msg.toUpperCase());
}

let myFunc: FunctionType = showMessage;
myFunc("Hello!"); // Output: Hello!
myFunc = showMessageUppercase;
myFunc("Hello!"); // Output: HELLO!
```

Example 3:

```
interface IStringList {
    [index: string]: string
}

let countries: IStringList = {};
countries["NL"] = "Netherlands";
countries["GR"] = "Greece";
console.log(countries["NL"]); // Output: Netherlands
console.log(countries["GR"]); // Output: Greece
console.log(countries); // Output: { NL: 'Netherlands', GR: 'Greece' }
```

Example 4:

```
interface Person {
    name: string;
}

interface Person {
    age: number;
}

// Interfaces with the same name are merged.
let person: Person;
person = { name: "John", age: 30 };
console.log(person.name, person.age);  // Output: John 30
```

Example 5:

```
interface Name {
    firstName: string;
}

interface FullName extends Name {
    lastName: string;
}

function printName(name: Name): void {
    console.log(name.firstName);
}

function printFullName(name: FullName): void {
    console.log(`${name.firstName} ${name.lastName}`);
}

printName({ firstName: "Kim" }); // Output: Kim
printFullName({ firstName: "Kim", lastName: "Wilde" }); // Output: Kim Wilde
```

Example 6:

```
interface FirstName {
    firstName: string;
}

interface LastName {
    lastName: string;
}

interface FullName extends FirstName, LastName {}

function printFullName(name: FullName): void {
    console.log(`${name.firstName} ${name.lastName}`);
}

printFullName({ firstName: "Kim", lastName: "Wilde" }); // Output: Kim Wilde
```

Classes

Example 1:

```
class Point {
    x: number;
    y: number;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }

    toString(): string {
        return `X: ${this.x}, Y: ${this.y}`;
    }
}

let myPoint1 = new Point(5, 10);
console.log(myPoint1.toString()); // Output: X: 5, Y: 10
let myPoint2 = new Point(2.5, -3);
console.log(myPoint2.toString()); // Output: X: 2.5, Y: -3
console.log(myPoint1.toString()); // Output: X: 5, Y: 10
myPoint1 = new Point(1, 2);
console.log(myPoint1.toString()); // Output: X: 1, Y: 2
```

Example 2:

```
// In TypeScript a class can not be static, but you can use an abstract
// class instead. You can also use a module.

abstract class MyMath {
    public static readonly PI: number = 3.14;

    public static twoPI(): number {
        return this.PI * 2;
    }
}

console.log(MyMath.PI); // Output: 3.14
console.log(MyMath.twoPI()); // Output: 6.28
```

Modules

Example 1:

myMath.ts:

```
// Compile this module: npx tsc --module CommonJS myMath.ts
export module MyMath {
    export let PI: number = 3.14;

    export function twoPI(): number {
        return PI * 2;
    }
}
```

myFile.ts:

```
import { MyMath } from "./myMath";

console.log(MyMath.PI); // Output: 3.14
console.log(MyMath.twoPI()); // Output: 6.28
MyMath.PI = 2;
console.log(MyMath.PI); // Output: 2
```

React types

Events
https://felixgerschau.com/react-typescript-events/

onChange	React.ChangeEvent<HTMLInputElement>
onClick	React.MouseEvent<HTMLButtonElement>
onKeyDown	React.KeyboardEvent<HTMLInputElement>
onSubmit	React.FormEvent<HTMLFormElement>

Example 1:

```
import React from 'react';

export default function Home(): React.ReactNode {
  return (
    <div>
      <h1>Home</h1>
      <p>This is the Home page</p>
    </div>
  );
}
```

Example 2:

```
import React from 'react';

export default function Home(): React.ReactNode {
  const [value, setValue] = React.useState('');

  function handleChange(e: React.ChangeEvent<HTMLInputElement>): void {
    setValue(e.target.value);
  }

  return (
    <div>
      <p>Enter your name:</p>
      <input value={value} onChange={handleChange} id="my-input" />
      <p>Hello {value}</p>
    </div>
  );
}
```

Example 3:

```
import React from 'react';

export default function Subscribe(): React.ReactNode {
  const [email, setEmail] = React.useState('');

  function handleChange(e: React.ChangeEvent<HTMLInputElement>): void {
    setEmail(e.target.value);
  }

  const handleSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    alert(email); // Replace this line
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <div>
          <label htmlFor="email">Email address</label>
        </div>
        <div>
          <input type="email" value={email} onChange={handleChange} id="email" />
        </div>
        <div>
          <button type="submit">Subscribe</button>
        </div>
      </form>
    </div>
  );
}
```

Example 4:

```
// Explicitly specifying the type for a useState hook to avoid (string | string[])[]

const [value, setValue] = useState<[string, string[], string[]]>(["", [""], [""]]);
```

Optional call

Example 1:

```
function getCurrentDateTimeAsString(): string {
    const current = new Date();
    const day = ('0' + current.getDate().toString()).slice(-2);
    const month = ('0' + (current.getMonth() + 1).toString()).slice(-2);
    const year = current.getFullYear().toString();
    const hours = ('0' + current.getHours().toString()).slice(-2);
    const minutes = ('0' + current.getMinutes().toString()).slice(-2);
    const seconds = ('0' + current.getSeconds().toString()).slice(-2);
    return `${day}-${month}-${year} ${hours}:${minutes}:${seconds}`;
}

function log(msg: string): void {
    console.log(`${getCurrentDateTimeAsString()} ${msg}`);
}

function error(msg: string, log?: (msg: string) => void): void {
    console.log(msg);
    log?.(msg);
}

error("This is an error!");
error("This is an error that needs to be logged!", log);
```

Links

https://www.typescriptlang.org/
https://www.tutorialsteacher.com/typescript
https://www.tutorialspoint.com/typescript/index.htm
https://github.com/Microsoft/TypeScript/wiki/FAQ
https://mariusschulz.com/blog/series/typescript-evolution
https://www.totaltypescript.com/concepts/property-does-not-exist-on-type
https://www.zhenghao.io/posts/ts-never

Converting to TypeScript
https://www.sitepoint.com/how-to-migrate-a-react-app-to-typescript/
https://www.sitepoint.com/react-with-typescript-best-practices/

Zustand
https://docs.pmnd.rs/zustand/guides/typescript