4 TypeScript

CS 425
Web Applications Development

Alex Petrovici

Content

Introduction /

What is TypeScript ?

- A superset of JavaScript. All valid JS code is valid TS code, but TS adds static typing and advanced tooling.
- TS code is compiled (transpiled) into plain JavaScript, which is what browsers actually execute.
- Created by Microsoft and first released in 2012, led by Anders Hejlsberg (also creator of C# and Turbo Pascal).
- Provides optional static types, interfaces, generics, enums, and modern ECMAScript features before browsers support them.
- Integrates tightly with modern editors (like VS Code) for autocompletion, refactoring, and type checking.

Introduction /

Why is it useful to know TS ?

- Core frameworks such as Angular are written entirely in TypeScript.
- Helps prevent runtime errors by catching type-related issues during development.
- Makes large codebases easier to maintain, refactor, and scale over time.
- Increasingly used in full-stack environments (Node.js, Deno, Next.js, etc.).

Introduction /

How to start with TS ?

Install npm (Node's ecosystem package manager)

```
sudo apt update && sudo apt install -y npm
```

Then we can install TypeScript:

```
npm install -g typescript
```

JavaScript foundations / Scope

Execution model (scope, closures, hoisting), ‘this‘ binding, prototypes, async/await, promises, and ES modules. Without that, TypeScript’s compile-time guarantees are meaningless.

JavaScript foundations / Scope

Global scope

```javascript
var globalVar = "I'm global";
let globalLet = "I'm also global";
const globalConst = "I'm global too";

console.log("=== Global Scope ===");
console.log(globalVar); // Accessible anywhere
```

JavaScript foundations / Scope

Function scope

```
function functionScopeExample() {
    var functionVar = "I'm function-scoped";

    if (true) {
      var insideIf = "var ignores block scope";
      console.log(functionVar); // Accessible
    }

    console.log(insideIf); // Still accessible! (var is function-scoped)
  }

  console.log("\n=== Function Scope ===");
  functionScopeExample();
  // console.log(functionVar); // Error: not accessible outside function
```

JavaScript foundations / Scope

Block Scope

```
function blockScopeExample() {
    console.log("\n=== Block Scope ===");
    if (true) {
      let blockLet = "I'm block-scoped";
      const blockConst = "I'm also block-scoped";
      var functionScoped = "I'm function-scoped";
      console.log(blockLet); // Accessible inside block
      console.log(blockConst); // Accessible inside block
    }
    // console.log(blockLet); // Error: not accessible outside block
    // console.log(blockConst); // Error: not accessible outside block
    console.log(functionScoped); // Accessible (var is function-scoped)
    // Block scope in loops
    for (let i = 0; i < 3; i++) {
      // i is block-scoped to this loop
    }
    // console.log(i); // Error: i is not defined
    for (var j = 0; j < 3; j++) {
      // j is function-scoped
    }
    console.log(j); // 3 (accessible because var is function-scoped)
  }
```

JavaScript foundations / Scope

Closures

```
function outerFunction() {
    const outerVar = "I'm from outer function";

    function innerFunction() {
      console.log(outerVar); // Can access outer function's variables
    }

    return innerFunction;
  }

  console.log("\n=== Lexical Scope (Closures) ===");
  const closure = outerFunction();
  closure(); // Still has access to outerVar
```

JavaScript foundations / Scope

Hoisting

Hoisting is JavaScript's behavior of moving declarations to the top of their scope before code execution.
Note: Relying on hoisting is a bad practice. It can lead to confusing code and subtle bugs. Always declare variables at the top of their scope and functions before use.

- `var` declarations are hoisted and initialized with `undefined`
- `let` and `const` are hoisted but not initialized (temporal dead zone)
- Function declarations are fully hoisted (both declaration and definition)
- Function expressions are not hoisted

JavaScript foundations / Scope

Hoisting

```
console.log(hoistedVar); // undefined (var is hoisted but not initialized)
// console.log(hoistedLet); // Error: Cannot access before initialization
// console.log(hoistedConst); // Error: Cannot access before initialization
var hoistedVar = "I'm hoisted";
let hoistedLet = "I'm not hoisted";
const hoistedConst = "I'm not hoisted either";

// Function declarations are fully hoisted
hoistedFunction(); // Works!
function hoistedFunction() {
  console.log("Function declarations are hoisted");
}

// Function expressions are not hoisted
// notHoisted(); // Error: notHoisted is not a function
var notHoisted = function () {
  console.log("Function expressions are not hoisted");
};
```

JavaScript foundations / Primitive data types

JavaScript types

- Primitive **types** in JavaScript are **lowercase**.
- The constructors are in **PascalCase** names refer to constructors (`new Number(5)`) or custom classes (`class Foo`).
- JavaScript has **runtime** primitive types:

  string, **number**, **boolean**, **null**, **undefined**, **symbol**, and **bigint**.

JavaScript foundations / Primitive data types

TypeScript types

TypeScript adds a **type system** on top:

- **any**: Disables type checking (avoid when possible).
- **unknown**: Type-safe alternative to `any`; requires type checking before use.
- **never**: Represents values that never occur (e.g., functions that always throw).
- **void**: Represents absence of a return value (functions that return nothing).
- **null** and **undefined** can be explicitly typed, whereas in JavaScript they are just values.
- Literal types (‘"on"‘, ‘42‘, ‘true‘) exist in TypeScript for exact value constraints.

At runtime, TypeScript compiles to plain JavaScript, so only JavaScript’s primitives remain.

JavaScript foundations / Structural typing

Classes vs Interfaces in JavaScript/TypeScript

- **typeScript**: Only has **classes** (ES6+). No native interface concept.
- **TypeScript**: Has both **classes** and **interfaces**.
- **Class**: Creates both a type and a runtime value (constructor function).
- **Interface**: Compile-time only; defines shape/contract but produces no JavaScript code.
- **Key difference**: Classes exist at runtime; interfaces are erased during compilation.
- Use **interfaces** for type contracts; use **classes** when you need instances/inheritance.

TypeScript is **structurally typed** or **duck typing**. Compatibility depends on shape, not declared name. Two different interfaces with the same fields are **interchangeable**.

JavaScript foundations / Structural typing

Object type inference

You can run the example files with the following commands:

```
# Compile TypeScript to JavaScript
npx tsc structural-typing.ts

# Run the compiled JavaScript
node structural-typing.js
```

or simply using:

```
npx tsx structural-typing.ts
```

npx is a tool that runs commands without installing them globally. **tsx** is a tool that runs TypeScript files directly without manually compiling them to JavaScript.

JavaScript foundations / Type inference

Basic structural compatibility

TypeScript infers types automatically from values and function returns. Make as few as possible, as many as necessary.

```
interface Point2D {
    x: number;
    y: number;
  }
  interface Vector2D {
    x: number;
    y: number;
  }
  // Even though Point2D and Vector2D are different interfaces,
  // they are structurally identical and thus interchangeable
  const point: Point2D = { x: 10, y: 20 };
  const vector: Vector2D = point; // No error! Same structure

  console.log("Point:", point);
  console.log("Vector (assigned from point):", vector);
```

JavaScript foundations / Unions and intersections

Union (‘|‘) for multiple possible types. Intersection (‘&‘) for combining constraints. These are the core of expressive type definitions.
Used to represent exact values and finite state machines safely.

```
npx tsc unions.ts
node unions.js
```

Basic union type

```
type StringOrNumber = string | number;

let value1: StringOrNumber = "hello";
console.log("value1 (string):", value1);

value1 = 42;
console.log("value1 (number):", value1);
```

JavaScript foundations / Unions and intersections

Function with union parameter

```
function printId(id: string | number) {
console.log("\nYour ID is:", id);

// Type narrowing with typeof
if (typeof id === "string") {
    console.log("ID is a string, uppercase:", id.toUpperCase());
} else {
    console.log("ID is a number, doubled:", id * 2);
}
}

console.log("\n=== Function with Union Parameter ===");
printId("ABC123");
printId(456);
```

JavaScript foundations / Unions and intersections

Union with literal types

```
type Status = "success" | "error" | "pending";

function handleStatus(status: Status) {
  switch (status) {
    case "success":
      console.log("Operation successful!");
      break;
    case "error":
      console.log("Operation failed!");
      break;
    case "pending":
      console.log("Operation pending...");
      break;
  }
}
handleStatus("success");
handleStatus("error");
handleStatus("pending");
```

JavaScript foundations / Unions and intersections

Union with objects

```
interface Dog {
type: "dog";
bark(): void;
}
interface Cat {
type: "cat";
meow(): void;
}
type Pet = Dog | Cat;
function makeSound(pet: Pet) {
    // Discriminated union - using 'type' property to narrow
    if (pet.type === "dog") {
      pet.bark();
    } else {
      pet.meow();
    }
  }
```

JavaScript foundations / Unions and intersections

Array of union types

```
const mixedArray: (string | number | boolean)[] = [
"hello",
42,
true,
"world",
100,
false,
];

console.log("\n=== Array of Union Types ===");
console.log("Mixed array:", mixedArray);

mixedArray.forEach((item) => {
console.log(`Value: ${item}, Type: ${typeof item}`);
});
```

JavaScript foundations / Generics

Parameterize types (‘Array<T>‘, ‘Promise<T>‘).

Basic generic function

```
function identity<T>(arg: T): T {
    return arg;
}

const stringResult = identity("hello");
const numberResult = identity(42);
const boolResult = identity(true);
```

- **<T>** declares a type parameter (generic)
- **arg: T** means the parameter has type **T**
- **: T** after the parentheses is the return type
- TypeScript infers **T** from the argument passed

JavaScript foundations / Narrowing and type guards

Runtime checks (‘typeof‘, ‘instanceof‘, custom predicates) that refine types within a block. Essential for safe code flow.

Type narrowing with `typeof` and `instanceof`

```
class Dog {
  bark() { console.log("Woof!"); }
}
class Cat {
  meow() { console.log("Meow!"); }
}

function makeSound(animal: Dog | Cat) {
  if (animal instanceof Dog) {
    animal.bark();
  } else {
    animal.meow();
  }
}
```