TypeScript Essentials

Eduardo Araujo

Summary

- Personal Presentation
- TypeScript - Why?
  - JavaScript Timeline
    - From ES5 to ES6
    - A new reality for JS
- TypeScript begins
  - Main Features
    - Type annotations
    - Interfaces
    - Enums
    - Generics
- TypeScript from scratch
  - Creating a TS project
  - package.json
  - tsconfig.json
- Proof of Concept

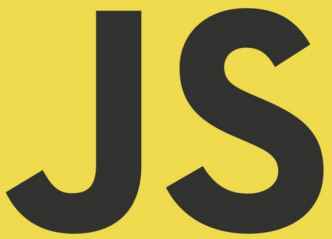
Eduardo Araujo

- Bachelor in Industrial Engineering
  - Universidade Federal Fluminense Niterói, Rio de Janeiro, Brazil
- 15+ years of IT environment
  - Self-taught developer (JavaScript)
  - University's IT new technologies development
  - IT Analysis
  - Incident Management
  - Full-stack Software Development
  - Data Engineering
    - Data Governance
    - Visibility
- Currently working as Data Engineer at **Transporeon GmbH**, Ulm, Germany

- Development experience:
  - Python
  - JavaScript
    - TypeScript
    - Node.JS
    - React
  - PHP
  - C# .NET
- Data Engineering experience:
  - Google BigQuery
  - DBT
  - Lightdash
  - Databricks
- Cloud experience
  - AWS
  - Google Cloud Platform
  - Azure

TypeScript: Why?

JavaScript Timeline



ES5

ES6

1995 | 1997 | ... | ... 2009 | ... 2015

JavaScript is created

High-level programming language for browsers

"Java" was a popular term

ECMAScript standard

All browsers conform to this standard

Web 1.0

Client-side scripting

Web 2.0

Creation of JS frameworks for Web Applications

Ajax, JQuery...

Web 3.0

Server-side standalone JavaScript
Node.JS
npm package manager

Advanced client-side web frameworks
React, Vue, Angular,...

From ES5 to ES6

ES5

- "use strict"
- Array handling improvements
- String handling improvements
- JSON parse and stringify
- Getters and Setters in objects
- bind (borrow method from object)
- Ignoring trailing commas (JSON DOESN'T)

ES6

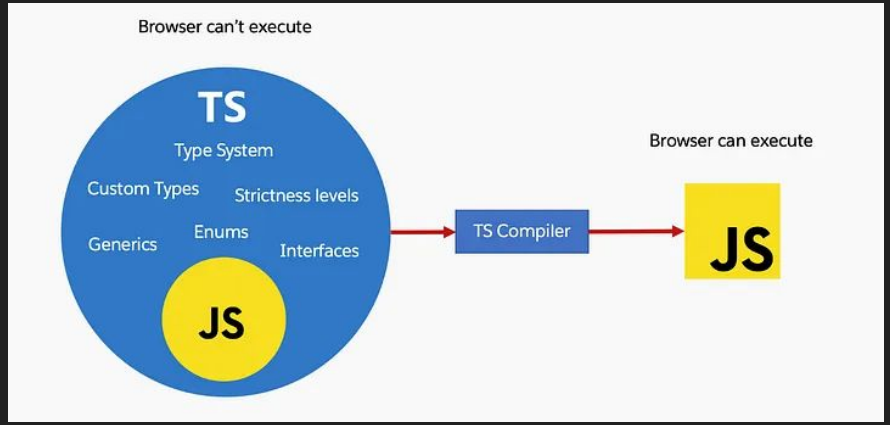
- Define variables with let and const
- Arrow functions
- Spread operators (...)
- Destructuring
- Maps (key-value list)
- Sets (unique elements list)
- Classes
- Promises (asynchronous objects)
- Default parameters
- Array handling improvements
- String handling improvements
- Import Modules!

A new reality for JS

- JavaScript was first thought for small client-side scripting
- Much simpler than Java, C++ and other languages
- Since Web 3.0 JavaScript has been used to build complex applications
- Simpler language in complex applications leads to more bugs
- How to make JavaScript development less buggy?
- To avoid bugs we need to validate code
- One of the forms of code validation is **type-checking**

TypeScript begins

- Released by Microsoft in 2012
- TS is a superset of JS
  - "All JavaScript code is TypeScript code”
- Essentially “JavaScript with types”
  - Also interfaces, enums, generics…
- TS code is compiled as JS code to be run
- The type-checking makes development and bug fixing faster
- Type-checking in compilation time, not runtime
- Currently TypeScript is
  - one of the Top 5 most liked languages
  - supported by many JS frameworks as React
  - used on many TypeScript-only frameworks as Nest.JS



Main TypeScript Features

Type annotations

TypeScript combines JS primitive types:

- `number`
  No difference between float and int in JS!
- `string`
- `bigint`
  New type for large integers; added in 2020.
- `symbol`
  Type for unique keys.
- `boolean`
- `null`
  Empty value. Not 0. Not “”. Just null.
- `undefined`
  No value assigned!
- `object`
  JS equivalent to record or struct

To additional types:

- `unknown`
  Use this when you don’t know the type.
- `any`
  “turns off” type checking, don’t overuse it.
- `never`
  Returns an error when any type is assigned.

- **object literal**
  `{property1: string, property2: number, …}`
- **mutable array**
  `string[]` or `Array<string>`
- **tuple**
  `[type1, type2, …]`
- **function**
  `(x: number, y: number) => number`
- **void**
  `(x: number) => void`

Type annotations

Some examples about how to use types:

```
let x: number;
x = 0; //OK
x = "hi" //ERROR

let y = 1;
y = 2; //OK
y = "hi" //ERROR, type inference

type listStr = string[];
const myList: listStr = ["a", "b", "c"]; //OK

/* Functions: each parameter has its type, and
also the return: */
function mySum(a: number, b: number): number {
      return a + b;
}
// void if the function returns nothing
function writeSomething(s: string): void {
      console.log(s);
}
```

```
// For arrow/lambda functions:
const wSub = (a: number, b: number): void => {
      console.log(a + b);
}

// Function as a parameter:
type myFunction = (s: string) => string

function runFunction(talkToMe: myFunction): void {
      console.log(talkToMe("John"));
}
runFunction(s => "Hello " + s)

// If type is unknown, you can cast it using "as".
let x: unknown = "hello";
console.log((x as string).length);
// "as" does not change the value, an error will
appear if casting does not work.
```

More about types

```
const myList: string[] = ["a"];
myList.push("b")
// No error; arrays can be changed if constant.

const myList: readonly string[] = ["a"];
myList.push("b") // ERROR; readonly arrays can't
be changed if constant.

/* Tuples are arrays in which each item can have a
different type */
let myTuple: [string, boolean, number];
myTuple = ["John", true, 31]; //OK
myTuple = ["John", "Doe", 31]; //ERROR

// Object literals describe each object element
type person = {name: string, age: number};
const p1: person = {name: "John", age: 31}; //OK
const p2: person = {name: "John", age: "31"};
//ERROR
//As arrays, they can be changed if constant.
```

```
// Types can be united
type Loading = {
 state: "loading";
};

type Failed = {
 state: "failed";
 code: number;
};

type Success = {
 state: "success";
 response: {
   duration: number;
   summary: string;
 };
};
/* Create a type representing only one of the
above types but you aren't sure which it is. */
type Network = Loading | Failed | Success | null;
```

Interfaces

Work almost interchangeably to object types:

```
type Failed = {
 state: string;
 code: number;
};

// is the same as:

interface Failed {
 state: string;
 code: number;
};
```

But…

```
/* Interfaces only work as object types. For
example the type below could never be an
interface. */
type MyList = string[];
/* Interfaces can be merged if declared more
than one time. Types would be overwritten */
interface Client {
    name: string;
}
interface Client {
    age: number;
}
const harry: Client = { name: 'Harry', age: 41
};
/* Interfaces can be extended. */
interface VipClient extends Client {
    benefits: string[];
}
```

Enums

A group of constant values. Good for statuses

```
enum StatusCodes {
  NotFound = 404,
  Success = 200,
  Accepted = 202,
  BadRequest = 400
}

console.log(StatusCodes.NotFound);// logs 404
console.log(StatusCodes.Success);// logs 200

// By default enums are numeric:
enum Activity {
  active,
  inactive,
  suspended
}
console.log(Activity.active);// logs 0
```

```
// If defined they can be strings as well:
enum Activity {
  active = "A",
  inactive = "I",
  suspended = "S"
}
console.log(Activity.active);// logs "A"
```

Generics

"Parameters for types"

```
type details<T> = {
  name: string,
  value: T,
  description: string
}

const ageDetails: details<number> = {
  name:"age",
  value: 31,
  description: "age of someone"
}

const usernameDetails: details<string> = {
  name:"username",
  value: "John",
  description: "username of someone"
}
```

```
/* Generics can be used to create generalized
functions */
function detailIt<T>(n: string, v: T, d:
string) : details<T> {
  return {
    name: n,
    value: v,
    description: d
  }
}
console.log(detailIt<number>("age", 31, "age
of someone"));
/* logs {
  name:"age",
  value: 31,
  description: "age of someone"
} */
```

TypeScript from scratch

Creating a TS project

1. Install and setup Node.JS
   (*Node.JS includes npm)*
2. Run `$ mkdir project_directory`
3. Run `$ cd project_directory`
4. Run `$ npm init -y`
   (*Creates `package.json` file)*
5. Run `$ npm i -D typescript ts-node`
   (Creates TS compiler and loader)
6. Run `$ tsc --init`
   (Creates `tsconfig.json` file)
7. Run `$ mkdir src`
8. Run `$ cd src`
9. Create a index.ts file with simple code
10. Run `$ cd ..`
11. Run `$ npx ts-node src/index.ts`

package.json

```
{
  "name": "project_name",
  "version": "1.0.0",
  "description": "Project description.",
  "main": "dist/index.js",
  "type": "module",
  "scripts": {
    "start": "npx ts-node src/index.ts",
    "test": "echo \"No test specified\" && exit 1"
  },
  "author": "Eduardo Araujo",
  "license": "ISC",
  "devDependencies": {
    "@types/express": "^4.17.20",
    "typescript": "^5.2.2"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Important properties to remind:

- main
  Must point to the starter JS compiled file
- type
  "module" allows ES6 imports. "commonjs" allows older require
- scripts
  Stores the useful commands that will be called during development and execution
- dependencies
  All packages that must be installed
- devDependencies
  Packages installed only if production environment flag is not set

tsconfig.json

```
{
  "compilerOptions": {
    "module": "nodenext",
    "esModuleInterop": true,
    "target": "es6",
    "noImplicitAny": true,
    "strictNullChecks": true,
    "removeComments": true,
    "sourceMap": true,
    "outDir": "dist"
  },
  "ts-node": {
    "esm": true
  },
  "lib": ["esnext"],
  "include": ["src"],
  "exclude": ["node_modules", "**/__tests__/*"]
}
```

Important properties to remind:

- module
  The TS compiler. You can compile TS for an old JS version for example. Recommended nodenext for new projects
- noImplicitAny
  Disables implicit "any" type
- removeComments
  Removes TS comments in JS compilation
- sourceMap
  Generates a map between TS code and JS code to help debuggers
- outDir
  Directory where the compiled JS files will be stored
- strictNullChecks
  Denies null and undefined values when they are not included as type
- target
  Language version of compiled output
- include
  Folders with files to compile
- exclude
  Folders with files to ignore
- ts-node
  Specific configurations for EcmaScript module running (essential for ES6)

Proof of Concept

Goal: A simple Twitter/X-like bulletin board or microblog.

We will need two TypeScript apps:

- Node.JS back-end with PostgresSQL database and read/write endpoints
- React.JS front-end to create new messages and see them.

They will communicate with each other using a simple REST API.

Dependencies:

- Postgres (pg node package)
- React
- Express.JS
- cors
- body-parser
- dotenv
- antd (Ant Design)

Do not forget the types of each package.
For quick front-end creation we will use `npm create vite@latest`