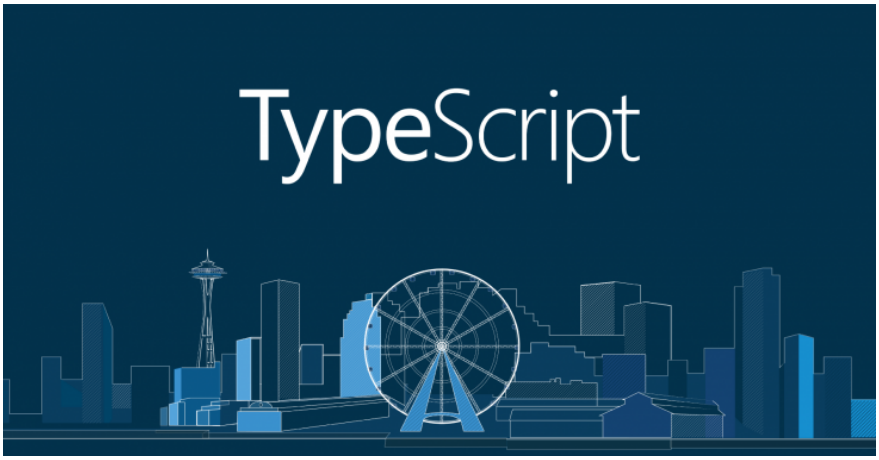
Questions and Answers

Vigo Webs Follow
Jun 3, 2018 · 7 min read



Q1. What is TypeScript? Why should we use it?

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript which runs on any browser or JavaScript engine.

TypeScript offers support for the latest JavaScript features and also has some additional features like static typing, object oriented programming and automatic assignment of constructor.

Q2. What are Types in TypeScript?

The type represents the type of the value we are using in our programs. TypeScript supports simplest units of data such as numbers, strings, boolean as well as additional types like enum, any, never.

In TypeScript, we are declaring a variable with its type explicitly by appending the `:` with the variable name followed by the type.

```
let decimal: number = 6;
let color: string = "blue";
let notSure: any = 4;
let unusable: void = undefined;
```

The reason adding types are:

- Types have proven ability to enhance code quality and understandability.
- It's better for the compiler to catch errors than to have things fail at runtime.
- Types are one of the best forms of documentation we can have.

Q3. What is Type assertions in TypeScript?

A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScript assumes that we have performed any special checks that we need.

```
let strLength: number = (someString).length;
```

Q4. What is `as` syntax in TypeScript?

The `as` is additional syntax for Type assertion in TypeScript. The reason for introducing the `as` -syntax is that the original syntax (`<type>`) conflicted with JSX.

```
let strLength: number = (someString as string).length;
```

When using TypeScript with `JSX` , only `as` -style assertions are allowed.

Q5. What is Compilation Context?

The compilation context is basically grouping of the files that TypeScript will parse and analyze to determine what is valid and what

isn't. Along with the information about which files, the compilation context contains information about *which compiler* options. A great way to define this logical grouping is using a `tsconfig.json` file.

A `tsconfig.json` might look like the following snippet:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "sourceMap": true,
    "outFile": "./",
    "rootDir": "./",
  }
}
```

Q6. Can an `interface` extends a `class` just like a `class` implements `interface`?

Yes, an `interface` extends a `class`, when it does it inherits the members of the class but not their implementations. Interfaces inherit even the private and protected members of a base class. This means that when you create an interface that extends a class with private or protected members, that interface type can only be implemented by that class or a subclass of it.

Q7. What are all the other access modifiers that TypeScript supports?

TypeScript supports access modifiers `public`, `private` and `protected` which determine the accessibility of a `class` member as given below:

- `public` - All the members of the class, its child classes, and the instance of the class can access.
- `protected` - All the members of the class and its child classes can access them. But the instance of the class can not access.
- `private` - Only the members of the class can access them.

If an access modifier is not specified it is implicitly `public` as that matches the convenient nature of JavaScript.

Also note that at runtime (in the generated JS) these have no significance but will give you compile time errors if you use them incorrectly.

Q8. What is Contextual typing?

TypeScript compiler can figure out the type if you have types on one side of the equation but not the other. For example, we can re-write the previous example function as follow:

```
let myAdd: (baseValue: number, increment: number) => number =
    function(x, y) { return x + y; };
```

Note that we can omit the typings on the right side of the equal since it can be figured out automatically by TypeScript. This helps cut down on the amount of effort to keep our program typed.

Q9. What is Generic Class?

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (`<>`) following the name of the class.

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber();
```

Q10. Explain Relative vs. Non-relative module imports.

Module imports are resolved differently based on whether the module reference is relative or non-relative.

A *relative import* is one that starts with `/` , `./` or `../` . Some examples include:

- `import Entry from "./components/Entry";`
- `import { DefaultHeaders } from "../constants/http";`

Any other import is considered non-relative. Some examples include:

- `import * as $ from "jquery";`
- `import { Component } from "@angular/core";`

A relative import is resolved relative to the importing file and cannot resolve to an ambient module declaration. We should use relative imports for our own modules that are guaranteed to maintain their relative location at runtime.

A non-relative import can be resolved relative to baseUrl, or through path mapping.

Q11. What is Triple-Slash Directive? What are some of the triple-slash directives?

Triple-slash directives are single-line comments containing a single XML tag. The contents of the comment are used as compiler directives.

Triple-slash directives are **only** valid at the top of their containing file. A triple-slash directive can only be preceded by single or multi-line comments, including other triple-slash directives. If they are encountered following a statement or a declaration they are treated as regular single-line comments, and hold no special meaning. Below are some of the triple-slash directives in TypeScript:

- The `/// <reference path="..." />` directive is the most common of this group. It serves as a declaration of dependency between files. Triple-slash references instruct the compiler to include additional files in the compilation process. If the compiler flag `--noResolve` is specified, triple-slash references are ignored; they neither result in adding new files, nor change the order of the files provided.
- Similar to a `/// <reference path="..." />` directive, this directive serves as a declaration of dependency; a `/// <reference types="..." />` directive, however, declares a dependency on a package. For example, including `/// <reference types="node" />` in a declaration file declares that this file uses names declared in `@types/node/index.d.ts`; and thus, this package needs to be included in the compilation along with the declaration file.

Q12. What is JSX? Can we use JSX in TypeScript?

JSX is an embeddable XML-like syntax. It is meant to be transformed into valid JavaScript. JSX came to popularity with the React framework. TypeScript supports embedding, type checking, and compiling JSX directly into JavaScript.

In order to use JSX in our file: we must name our file with a `.tsx` extension and should enable `jsx` option.

Q13. What are all the JSX modes TypeScript supports?

TypeScript ships with three JSX modes: `preserve`, `react`, and `react-native`.

The `preserve` mode will keep the JSX as part of the output to be further consumed by another transform step (e.g. *Babel*). Additionally the output will have a `.jsx` file extension. The `react` mode will emit `React.createElement`, does not need to go through a JSX transformation before use, and the output will have a `.js` file extension. The `react-native` mode is the equivalent of preserve in that it keeps all JSX, but the output will instead have a `.js` file extension.

Q14. Why TypeScript is referred as Optionally Statically Typed Language?

TypeScript is referred as optionally statically typed, which means we can make the compiler to ignore the type of a variable optionally. Using `any` data type, we can assign any type of value to the variable. TypeScript will not give any error checking during compilation.

```
var unknownType: any = 4;
unknownType = "Okay, I am a string";
unknownType = false; // set a boolean. No error thrown at compile-time.
```

Q15. What is TypeScript Definition Manager?

When using TypeScript, you will need TypeScript definition files to work with external libraries. TypeScript Definition Manager (TSD) is a

package manager to search and install TypeScript definition files directly from the community driven **DefinitelyTyped** repository.

Consider we need typings file for `jQuery` so that we can use jQuery with TypeScript. This command, `tsd query jquery --action install` (we need to have `tsd` installed), finds and install the typings file for `jQuery`. Now we can include the below directive at the top of the file where we want to use `jQuery`.

```
/// <reference path="typings/jquery/jquery.d.ts" />
```

TSD is now offically deprecated, and we should use `typings` instead.

For more TypeScript Interview Questions and Answers visit my blog: http://blog.vigowebs.com/post/2018/typescript-interview-questions/