TypeScript

Context

- Before ES6, writing large applications in JavaScript is difficult, not originally designed for large complex applications(mostly a scripting language, with functional programming constructs).
- Lacks structuring mechanisms: Class, Module, Interface.

Different Approaches to Fix/Improve JS

- Through Library or Frameworks
  - jQuery, AngularJS, Knockout, Ext JS…
- New Language that extend/improve language features of JavaScript. Superset of JS, compiles to JS.
  - CoffeeScript, TypeScript
- Entirely new language with many new features that compile to JS
  - GWT(Google Web Toolkit), Dart

https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

TypeScript

- Open Source
- Adds additional features like Static Type(Optional), Class, Module, etc
- TypeScript is JavaScript. Any valid .js file can be renamed to .ts and compiled with other TypeScript files.
- Runs on Any browser, Any host, Any OS.
- TypeScript purposefully borrows ideas from ES6 spec – Class, Module

Features

- Optional Static Type Annotation/Static Typing
- Additional Features for Functions
  - Types for function parameters and return type, optional and default parameter, rest parameter, overloading
- Class
  - Field, Property, Method, Constructor, Event, Static methods, inheritance
- Interface
- Module
- Generics
- Declaration Merging
- Few Other Features (Enum)
- TypeScript comes with
  - TypeScript Compiler (tsc)
  - TypeScript Language Service (TLS) / VSC extension
  - Playground
  - Declaration files (*.d.ts) for DOM, jQuery, node.js
  - Language Spec and code examples

Install TypeScript

To install the TypeScript compiler:

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

```
class Person {}
```

app.ts

```
tsc app.ts //will create app.js
```

(It will convert the class defined to what a class would be defined as in ES5)

```
tsc -w app.ts //watching for file changes
```

tsconfig.json

- To create a TypeScript configuration file:

```
tsc -init //it creates a tsconfig.json
```

- Since all configurations live in this file we could simply type **`tsc`** it's going to automatically find all **`*.ts`** and compile them to JavaScript.

```
function log(someArg) { // error
  console.log(someArg);
}

log(123);
log('hello world');

let y: any; // explicit any is allowed
```

Useful for debugging in browser!

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es6",
        "noImplicitAny": false,
        "sourceMap": false
    },
    "exclude": [ "node_modules" ]
}
```

tsconfig.json

https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

Types/Optional Type Annotation

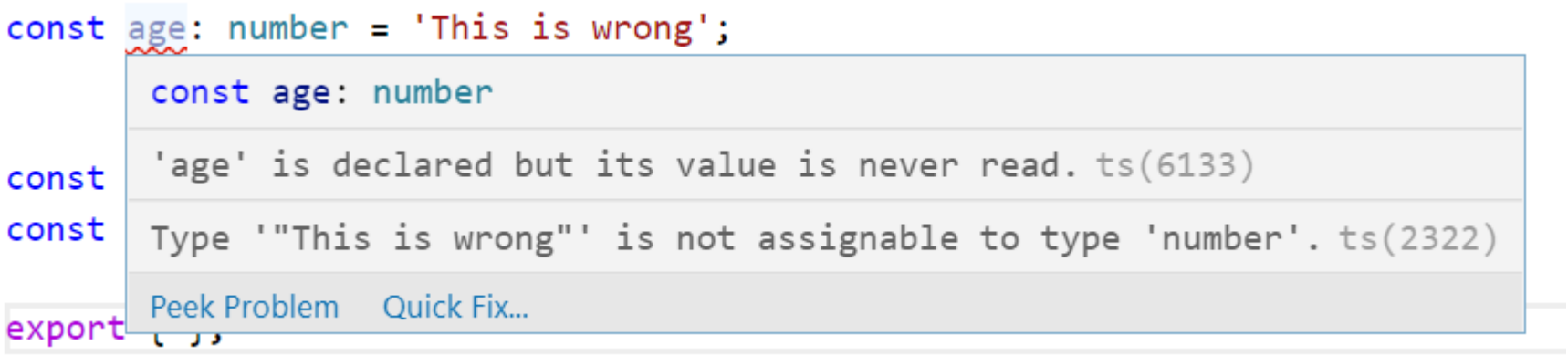
Optional Type Annotation

- TypeScript allows annotating variables with types
- No additional code is emitted in the final JavaScript that TypeScript Compiler produces

```typescript
const age: number = 40;
const name: string = 'TypeScript';
const isCheck: boolean = false;
```

```javascript
var age = 40;
var name = 'TypeScript';
var isCheck = false;
```

- If there's a type mismatch TypeScript shows a warning.



Basic Types

- Primitive
  - Number
  - Boolean
  - String
  - Null
  - Undefined
- Array
- Enum
- Tuple
- Any
- Void
- Never
- Object

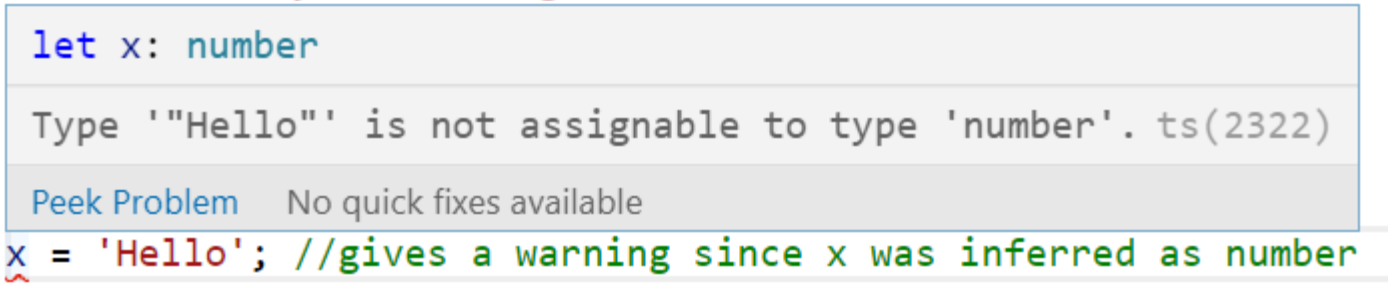
https://www.typescriptlang.org/docs/handbook/basic-types.html#number

DataTypes

- Number - Does not have separate integer and float/double type
  - All numbers in TypeScript are floating point values. These floating point numbers get the type `number`.
  - `let decimal: number = 6;`
  - `let hex: number = 0xf00d;`
  - `let binary: number = 0b1010;`
  - `let octal: number = 0o744;`
- Boolean – true/false value
  - `let isDone: boolean = false;`
- String – Both Single quote or double quote could be used
  - `let color: string = "blue";`
  - `color = 'red';`
- No Separate char type
  - `const character = 'a';`

Optional Type Annotation

- TypeScript tries to infer type
  - `let x = 21; //type inferred as number`
  - `x = 'Hello'; //gives a warning since x was inferred as number`



Type Inference

- Four ways of variable declaration
  - Declare its type and value (as a literal) in one statement
  - Declare its type but no value. The value will be set to undefined
  - Declare its value but no type. The variable will be of type Any, but its type may be inferred based on its value
  - Declare neither value nor type. The variable will be of type Any, and its value will be undefined.

```
// Option 1: Declare type and value
let message1: string = 'Hello from TypeScript';

// Option 2: Declare variable type, but no value
let message2: string;
// value could be assigned later
message2 = 'Hello from TypeScript';

// Option 3: Declare the value but no type
// TS try to infer the value
let message3 = 'Hello from TypeScript';

// Option 4: Neither declare type or value
// Type is inferred as Any - value of any type can be assigned
let message4;
```

Array

- TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by `[]` to denote an array of that element type:
  - `let list1: number[] = [1, 2, 3];`

- The second way uses a generic array type, `Array<elemType>`:
  - `let list2: Array<number> = [1, 2, 3];`

Enum

- A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.
  - ```
    enum Color { Red, Green, Blue }
    ```
  - ```
    let c: Color = Color.Green;
    ```

- By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:
  - ```
    enum Color { Red = 1, Green, Blue }
    ```
  - ```
    let c: Color = Color.Green;
    ```

  - ```
    let colorName: string = Color[2];
    ```
  - ```
    console.log(colorName); // Displays 'Green' as its value is 2 above
    ```

Any

- describe the type of variables that we do not know when we are writing an application
- May come from dynamic content, e.g. from the user or a 3rd party library
- Allows to opt-out of type-checking and let the values pass through compile-time checks
- Same as not declaring any datatype – uses JavaScript's dynamic nature

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

let list: any[] = [1, true, "free"];
list[1] = 100;
```

Void

- `void` is a little like the opposite of `any`: the absence of having any type at all.
- commonly see this as the return type of functions that do not return a value

```
function warnUser(): void {
    console.log("This is my warning message");
}
```

Funtion

Function Overview

- Functions are the fundamental building block of any applications in JavaScript.
- Allows build up layers of abstraction, mimicking classes, information hiding, and modules
- In TypeScript, while there are classes, namespaces, and modules, functions still play the key role in describing how to *do* things.
- TypeScript also adds some new capabilities to the standard JavaScript functions to make them easier to work with.
  - Type Annotation for parameter and return type
  - Optional and Default Parameter
  - Rest Parameter
  - Function Overloads

Function

- Typing the function: allows parameter and return type annotation

```
function add(x: number, y: number): number {
    return x + y;
}

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

Function Overloads

- Allows function overloads

```
function add(a: string, b: string): string;

function add(a: number, b: number): number;

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

add("Hello ", "Steve"); // returns "Hello Steve"
add(10, 20); // returns 30    return var1 + var2;
```

Optional & Default Parameter

- Optional Parameters should have default value that would be used if the value is not specified while invoking the function

```
function displayMsg(msg: string, user: string = 'default user') {
    console.log(msg + ' ' + user);
}
displayMsg('Hello');
displayMsg('Hello', 'TypeScript’);
```

- Should be the last arguments in a function

```
function displayMsg2(s1: string, s2: string = 'msg2', s3: string = 'msg3'){}

displayMsg2('first');
displayMsg2('first', 'second');
displayMsg2('first', 'second', 'third');
displayMsg2('first', , 'third') //doesn't work
```

Class

Class

- Properties and fields to store data
- Methods to define behavior
- Events to provide interactions between different objects and classes

Field/Property/Method/Constructor

```
class Employee {
    name: string;
    private _salary: number;

    get salary() {
        return this._salary;
    }

    set salary(value: number) {
        if (value <= 0) {
            throw 'Salary cannot be less than 0';
        } else {
            this._salary = value;
        }
```

```
const emp: Employee = new Employee();

try {
    emp.salary = -100;
} catch (err) {
    console.log(err);
}

emp.salary = 500;
console.log(emp.salary);
```

Constructor

- Uses constructor keyword
- public by default, cannot be private

```
class Employee {
    private name: string;
    private basic: number;
    private allowance: number;

    constructor(name: string, basic: number, allowance: number) {
        this.name = name;
        this.basic = basic;
        this.allowance = allowance;
    }
}
```

```
class Employee {

    constructor(name: string, basic: number, allowance: number) {
    }
}
```

Access Modifiers

- public(default) – member is available to all code in another module
- private – member is available only to other code in the same assembly

```
class Employee {
    public public_method() {
        console.log('public method called');
    }

    private private_method() {
        console.log('private method called');
    }
}

cons
emp.
emp.private_method();
```

```
(method) Employee.private_method(): void

Property 'private_method' is private and only accessible within class
'Employee'. ts(2341)

Peek Problem    No quick fixes available
```

Static Methods

- static methods are visible on the class itself rather than on the instances

```
class Employee {
    instanceMethod() {
        console.log('Employee instance method called.');
    }
    static staticMethod() {
        console.log('Employee static method called.');
    }
}

new Employee().instanceMethod();
Employee.staticMethod();
```

Inheritance

- TypeScript supports inheritance of class through `extends` keyword

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}
```

```
class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

// Slithering...
// Sammy the Python moved 5m.
sam.move();
// Galloping...
// Tommy the Palomino moved 34m.
tom.move(34);
```

Module

Module

- Modules are executed within their own scope, not in the global scope
- Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.
- In TypeScript, just as in ECMAScript 2015, any file containing a top-level import or export is considered a module.
- Conversely, a file without any top-level `import` or `export` declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Module

- Exporting a declaration
  - Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the export keyword.

StringValidator.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./StringValidator";

export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

  - Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

```
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

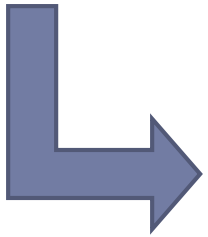
Interface & Decorators

Interface

- One of TypeScript's core principles is that type checking focuses on the *shape* that values have. This is sometimes called "duck typing" or "structural subtyping".

```
function printLabel(labeledObj: { label: string }) {
    console.log(labeledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```



```
interface LabeledValue {
    label: string;
}

function printLabel(labeledObj: LabeledValue) {
    console.log(labeledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Optional Properties

- Not all properties of an interface may be required.
- These optional properties are popular when creating patterns like "option bags" where you pass an object to a function that only has a couple of properties filled in.

```
interface SquareConfig {
    color?: string;
    width?: number;
}

                                                    return type
function createSquare(config: SquareConfig): { color: string; area: number } {
    let newSquare = { color: "white", area: 100 };
    if (config.color) {
        newSquare.color = config.color;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}

let mySquare = createSquare({ color: "black" });
```

Decorators

- Require additional features to support annotating or modifying classes and class members.
- Multiple decorators can be defined on the same Class/Property/Method/Parameter.
- Decorators are not allowed on constructors.
- Decorators are an **experimental** feature that may change in future releases.

- To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line or in your `tsconfig.json`:

- Command Line:

```
tsc --target ES5 --experimentalDecorators
```

- tsconfig.json:

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

Class Decorator

- A *Class Decorator* is declared just before a class declaration. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition.
  - If a decorator expression has the arguments e.g. @expression("a", 3) then these arguments are passed to the decorator function expression(arg1: string, arg2: number).
  - If a decorator expression has no arguments e.g. @expression then the constructor function of the class is passed to the decorator function expression(theConstructor: Function)
  - In both above cases the decorator expression can return a new constructor function to replace or override the original constructor function.

Class Decorator

Decorator Signature

ClassDecorator = <TFunction extends Function>(target: TFunction) => TFunction;

```
function SelfDriving(constructorFunction: Function) {
    console.log('-- decorator function invoked --');
    constructorFunction.prototype.selfDrivable = true;
}
```

```
function Wheels(numOfWheels: number) {
    console.log('-- decorator factory invoked --');
    return function (constructor: Function) {
        console.log('-- decorator invoked --');
        constructor.prototype.wheels = numOfWheels;
    }
}
```

```
@SelfDriving
@Wheels(5)
class Vechical {
    constructor(private make: string) {
        console.log('-- this constructor invoked --');
    }
}
```

```
console.log('-- creating an instance --');
let vechical: Vechical = new Vechical("Nissan");
console.log(vechical);

console.log(vechical['wheels']);
console.log(`${vechical['selfDrivable']}`);
```

Output in Terminal:

```
-- decorator factory invoked --
-- decorator invoked --
-- decorator function invoked --
-- creating an instance --
-- this constructor invoked --
Vechical { make: 'Nissan' }
-- decorator factory invoked --
-- decorator invoked --
-- decorator function invoked --
-- creating an instance --
-- this constructor invoked --
Vechical { make: 'Nissan' }
5
```

Method Decorator

- **Decorator Signature**
  - MethodDecorator = <T>(target: Object, key: string, descriptor: TypedPropertyDescriptor<T>) => TypedPropertyDescriptor<T> | Void;

  - **target**: Either the constructor function of the class for a static method, or the prototype of the class for an instance method.
  - **propertyKey**: The name of the method.
  - **descriptor**: The Property Descriptor for the method.

Method Decorator

```
function logMethod(target: Object, propertyName: string, propertyDesciptor: PropertyDescriptor): PropertyDescriptor {
    // target === Employee.prototype
    // propertyName === "greet"
    // propertyDesciptor === Object.getOwnPropertyDescriptor(Employee.prototype, "greet")
    const method = propertyDesciptor.value;

    propertyDesciptor.value = function (...args: any[]) {

        // convert list of greet arguments to string
        const params = args.map(a => JSON.stringify(a)).join();

        // invoke greet() and get its return value
        const result = method.apply(this, args);

        // convert result to string
        const r = JSON.stringify(result);

        // display in console the function call details
        console.log(`Call: ${propertyName}(${params}) => ${r}`);

        // return the result of invoking the method
        return 'result';
    }
    return propertyDesciptor;
}
```

```
class Employee {

    constructor(private firstName: string, private lastName: string) { }

    @logMethod
    greet(message: string): string {
        return `${this.firstName} ${this.lastName} says: ${message}`;
    }

}

const emp = new Employee('Josh', 'Edward');
const result = emp.greet('hello’);

//result: Josh Edward says: hello
console.log(`result: ${result}`);
```

Property Decorator

- **Decorator Signature**
  - PropertyDecorator = (target: Object, key: string) => void;
  - target — current object's prototype
    - i.e — If Employee is an Object, Employee.prototype
  - propertyKey — name of the property

Property Decorator

```
function logParameter(target: Object, propertyName: string) {
    // target -> Person.prototype
    //  propertyName: name

    // property value
    let _val = this[propertyName];

    // property getter method
    const getter = () => {
        console.log(`Get: ${propertyName} => ${_val}`);
        return _val;
    };

    // property setter method
    const setter = newVal => {
        console.log(`Set: ${propertyName} => ${newVal}`);
        _val = newVal;
    };

    // Delete property.
    if (delete this[propertyName]) {

        // Create new property with getter and setter
        Object.defineProperty(target, propertyName, {
            get: getter,
            set: setter,
            enumerable: true,
            configurable: true
        });
    }
}
```

```
class Person {

    @logParameter
    name: string;

}

const person = new Person();
person.name = 'Josh Edward';

// Set: name => Josh Edward
// Get: name => Josh Edward
// Josh Edward
console.log(person.name);
```

Parameter Decorator

- **Decorator Signature**
  - ParameterDecorator = (target: Object, propertyKey: string, parameterIndex: number) => void;
  - target — current object's prototype
    - i.e — If Employee is an Object, Employee.prototype
  - propertyKey — name of the method
  - index — position of the parameter in the argument array

- The return value from the decorator function is ignored.

Parameter Decorator

```
function logParameter(target: Object, propertyName: string, index: number)
 {
    // target: Student.prototype
    // propertyName: message
    // index: 0

    // generate metadatakey for the respective method
    // to hold the position of the decorated parameters
    const metadataKey = `log_${propertyName}_parameters`;
    console.log(metadataKey);
    if (Array.isArray(target[metadataKey])) {
        target[metadataKey].push(index);
    }
    else {
        target[metadataKey] = [index];
    }
}
```

```
class Student {
    greet(@logParameter message: string): string {
        return `hello ${message}`;
    }
}
const student = new Student();
student.greet('hello');
```

Resources

- TypeScript:
  - https://www.typescriptlang.org/

- Decorators:
  - https://codeburst.io/decorate-your-code-with-typescript-decorators-5be4a4ffecb4
  - https://www.logicbig.com/tutorials/misc/typescript/class-decorators.html