TypeScript Basics & Best Practices

Kevin Van Houtte

What's wrong with JavaScript

- Lack of modularity
- Dynamic Typing
- Verbose Patterns -> IIFE

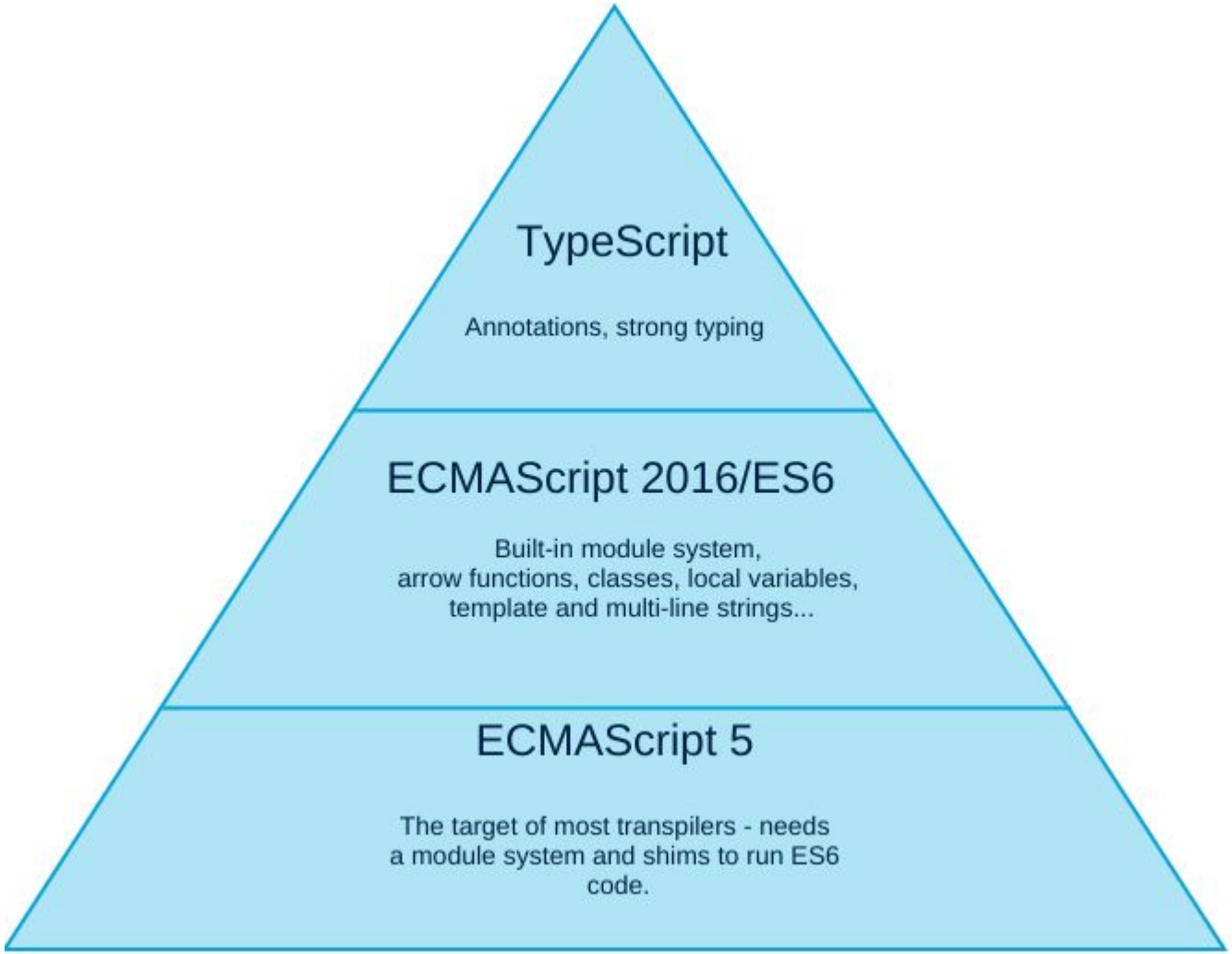
```
<script>
(function() {
    var name = "World";
    console.log('Hi ' + name);
})();
</script>
```

TypeScript Backstory

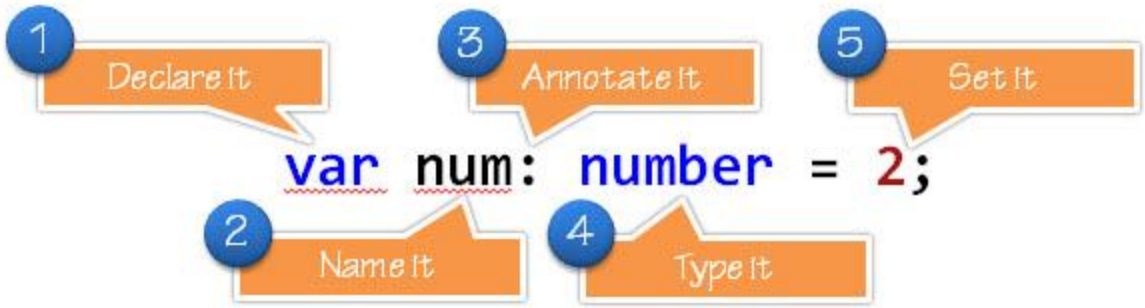
- Developed by Microsoft
- Open Source github.com/Microsoft/TypeScript
- Release: October 1 2012
- Latest stable release: 1.8 January 2016
- Conforms to ECMA standards & proposals

What is TypeScript?

- Superset of JavaScript
- Optionally typed
- Prevents common bugs and errors
- Readable code
- Transpiles to ES5/ES6
- OO design
- Module system
- Scalable for larger applications
- Long term



Types


1
Declare It
3
Annotate It
5
Set It
var num: number = 2;
2
Name It
4
Type It

Declare It!

- Let
- Var
- Const

Old days

```
var length = 16;                                    // Number
var lastName = "Johnson";                           // String
var cars = ["Saab", "Volvo", "BMW"];                // Array
var x = {firstName:"John", lastName:"Doe"};         // Object
```

Var (Function Scope)

Global	Inside of a function	Inside a function within a function

`var a = 10;`	`function f(){`
	`  var message = “Hello world!”;`
	`  return message;`
	`}`
		`function f(){`
		`  var a = 10;`
		`  return function g(){`
		`  var b = a + 1;`
		`  return b;`
		`  }`
		`}`

Scoping rules (var)

Variable recognition	Double trouble

function f(shouldInitialize: boolean){
  if (shouldInitialize) {
  var x = 10;
  }
  return x;
}
f(true); // returns '10'
f(false); // returns 'undefined'
	function sumMatrix(m:number[][]){
	var sum = 0;
	for (var i = 0; i <m.length; i++)
	{
	var cRow = matrix[i];
	for (var i = 0;i < cRow.length;
	i++) {
	  sum += currentRow[i];
	}}
	  return sum;
	}

Let (Block Scope)

- New way & Best practice
- Avoid common bugs like with var
- The declared variable only exists in the block scope (if, for, etc)

Scoping rules (let)

Variable recognition	Double trouble	Order of declaration

function f(input: boolean) {
  let a = 100;
  if (input) {
// Still okay to reference 'a'
  let b = a + 1;
  return b;
  }
  // Error: 'b' doesn't exist here
  return b;
}
	Not possible:
	let x = 10;
	let x = 20;
	 // error: can't re-declare 'x' in the same scope
	Possible:
	function f(condition, x) {
	  if (condition) {
	  let x = 100;
	  return x;}
	  return x;}
	
	f(false, 0); // returns '0'
	f(true, 0); // returns '100'
		a++
		let a;
		 // illegal to use 'a' before it's declared;

Const

- New way
- Same scoping rules as let
- You can’t reassign the variables

Type It!

- number
  - `let decimal: number = 5;`
- string
  - `let color: string = "red" or 'red';`
- boolean
  - `let isFinished: boolean = false;`
- array
  - `let list: number[] = [1,2,3]; or let list: Array<number> = [1,2,3];`
- tuple (fixed length)
  - `let x: [string,number] = [ "hello" , 5 ];`
- any
  - `let decimal: any = "notSureVariable";`
- void
  - having no type => used for methods that have no return value

Destructuring

Array destructuring

```
example 1
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2

example 2
let [first, ...rest] = [1, 2, 3,
4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2,
3, 4 ]
```

Object destructuring

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
}
let {a, b} = o;
```

Property renaming

```
let {a: newName1, b: newName2} = o;
```

With types

```
let {a, b}: {a: string, b: number}
= o;
```

Function declarations

```
type C = {a: string, b: number}
function f({a, b}: C): void {
    // ...
}
```

Default values

```
function keepWObject(wholeObject:
{a: string, b?: number}) {
    let {a, b = 1001} =
wholeObject;
}
```

Types @ Compile

JavaScript	TypeScript

`var a = 123`
`a.trim()`
 	
`TypeError: undefined is not a function`
	`var a: string = 123`
	`a.trim()`
	
	`Cannot convert 'number' to 'string'`

Module System

- Namespaces (internal modules)
- Optional hierarchical
- Export
- Import

Namespaces

- Previously “Internal modules”
- No name collisions
- No global variables
- Multi-file namespaces
- Optional Hierarchy (Submodules)

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;
}
namespace Validation.LettersOnlyValidation{}
```

Export & Import

- Viable options to export and import are with:
  - variables
  - functions
  - classes
  - interfaces
  - type aliases
- Wrap one or more modules to combine all exports
  - using export * from “module”
- Wrap all exports to a single variable using import

```
export interface StringValidator {}
export class Validator{}
export const regex= /^[0-9]+$/;
```

```
import { ZipCodeValidator } from "./ZipCodeValidator";
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

Classes

- Inheritance
- Can implement interfaces
- Members/methods (instance & static)
- Single constructor
- ES6 class syntax
- Access modifiers

Inheritance

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0):void {
        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);   }}

let sam = new Snake("Sammy the Python");
sam.move();
```

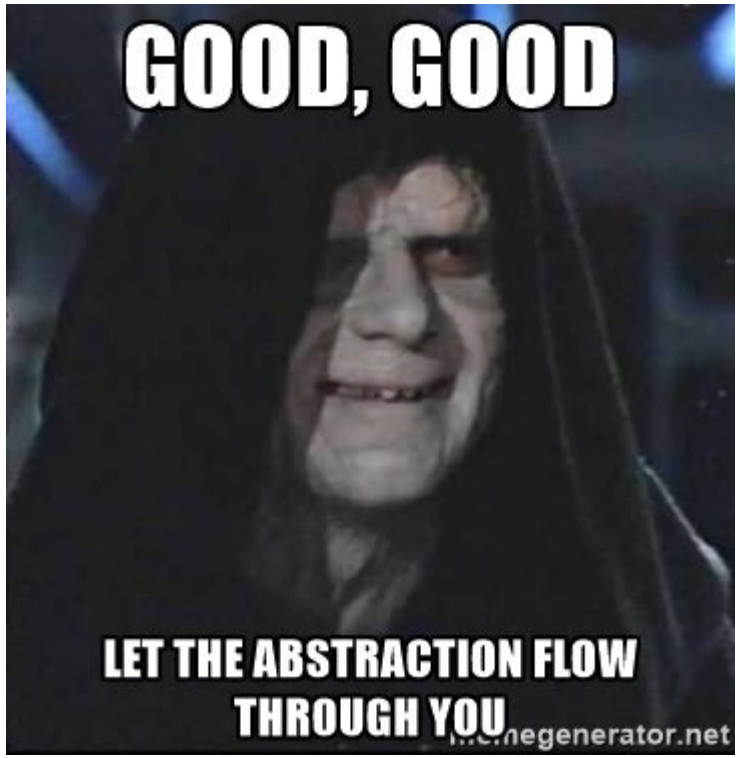
Access modifiers

- Public by default
- Private
- Protected
- Accessors

```
let passcode = "secret passcode";
class Employee {
    private _fullName: string;
    get fullName(): string {   return this._fullName;}
    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;        }
        else {
            console.log("Error: Unauthorized update of employee!");        }}}
                                let employee = new Employee();
                                employee.fullName = "Bob Smith";
                                if (employee.fullName) {
                                    console.log(employee.fullName);
                                }
```

Abstract Classes

- Prefix abstract
- Super class
- May not be initiated
- May contain implementation
- Abstract methods => must be implemented in the derived class
- Optionally include access modifiers



Interfaces

- Enforcing that a class meets a particular contract
- intent of the class
- Extends (multiple) interfaces

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;    }
 constructor(h: number, m: number) { }}
```

Tsconfig.json

- Specifies the root files and compiler options required to compile the project
- http://json.schemastore.org/tsconfig

TypeScript + AngularJS

- For better migration to Angular 2
- AngularJS is the mature framework
- All advantages from TypeScript
- All advantages from AngularJS

https://ordina-jworks.github.io/angularjs-typescript/2016/03/16/AngularTS.html


ITS OVER
ITS FINALLY OVER
quickmeme.com

Practice

https://github.com/KevinDaHub/Ordina-JWorks-TypeScript

Sources

https://www.typescriptlang.org/

https://github.com/Microsoft/TypeScript

https://github.com/Microsoft/TypeScript/wiki/Roadmap

https://github.com/KevinDaHub/Ordina-JWorks-TypeScript