TypeScript Fundamentals

21/10/2017

What is TypeScript?

"TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.“ ~ typescriptlang.org

JavaScript Dynamic Types

- **JavaScript provides a dynamic type system**

- **The Good:**
  - Variables can hold any object
  - Types determined on the fly
  - Implicit type coercion (ex: string to number)

- **The Bad:**
  - Difficult to ensure proper types are passed without tests
  - Not all developers use ===
  - Enterprise-scale apps can have 1000s of lines of code to maintain

Flexible Options

Any Browser

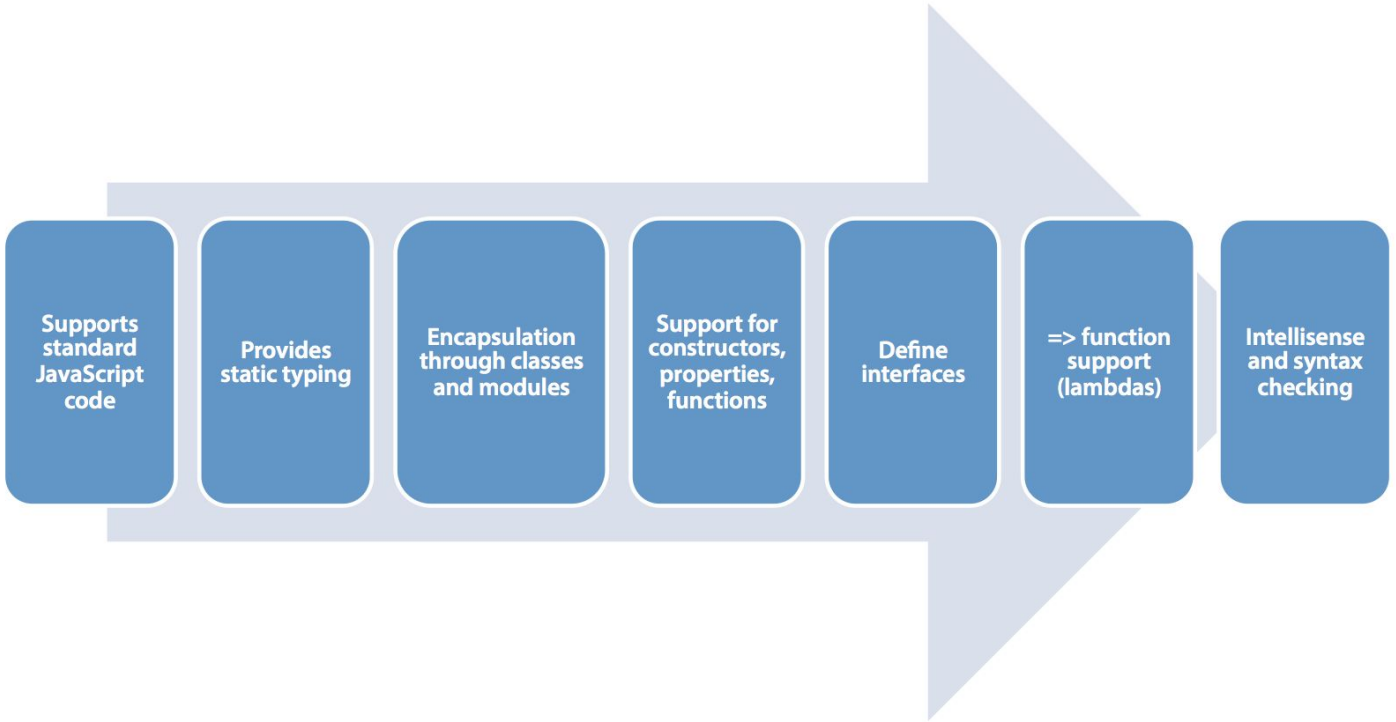
Any Host

Any OS

Open Source

Tool Support

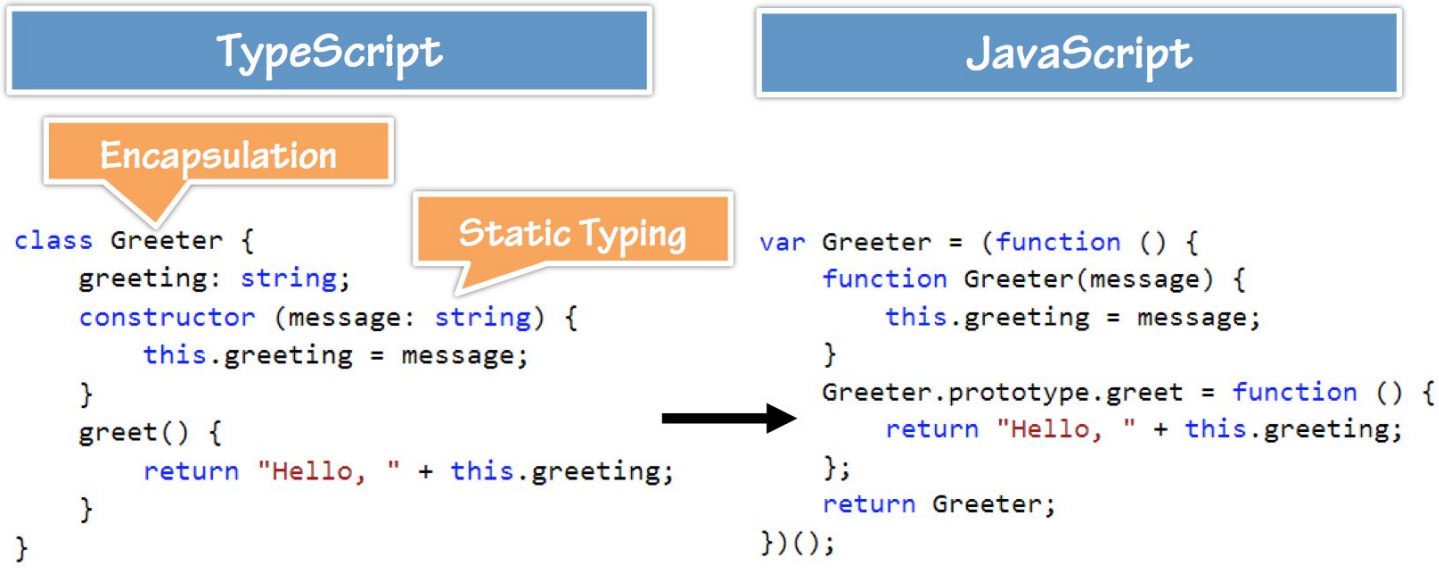
Key TypeScript Features



TypeScript Compiler

tsc first.ts

TypeScript → JavaScript



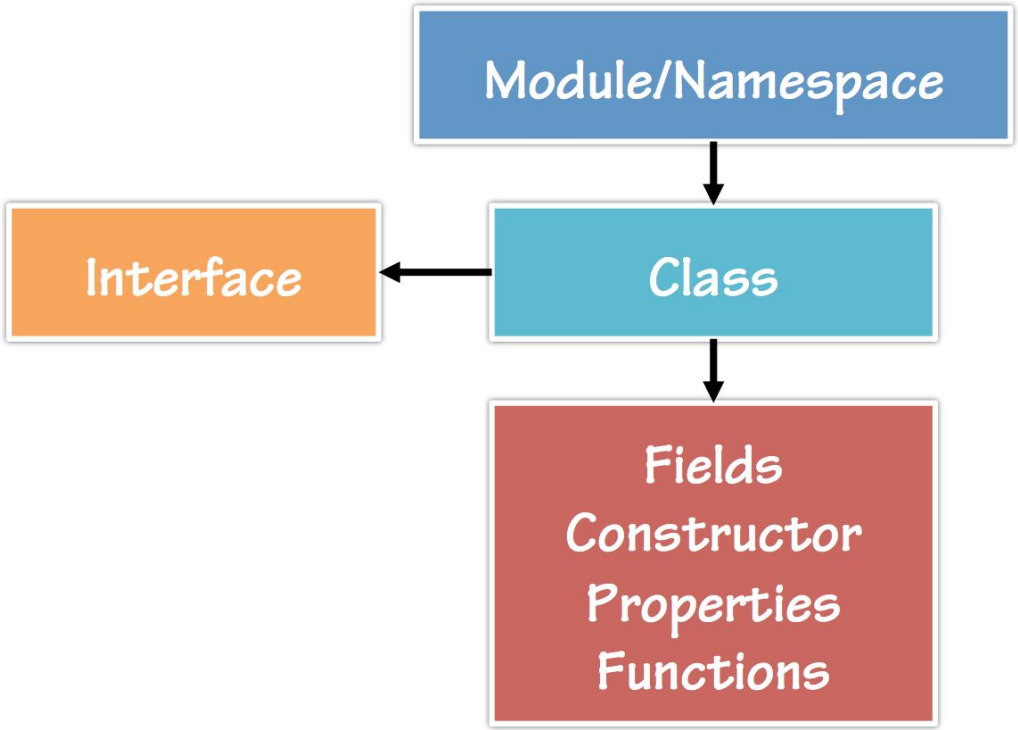
TypeScript Syntax Rules

- **TypeScript is a superset of JavaScript**
- **Follows the same syntax rules:**
  - { } brackets define code blocks
  - Semi-colons end code expressions
- **JavaScript keywords:**
  - for
  - if
  - More..

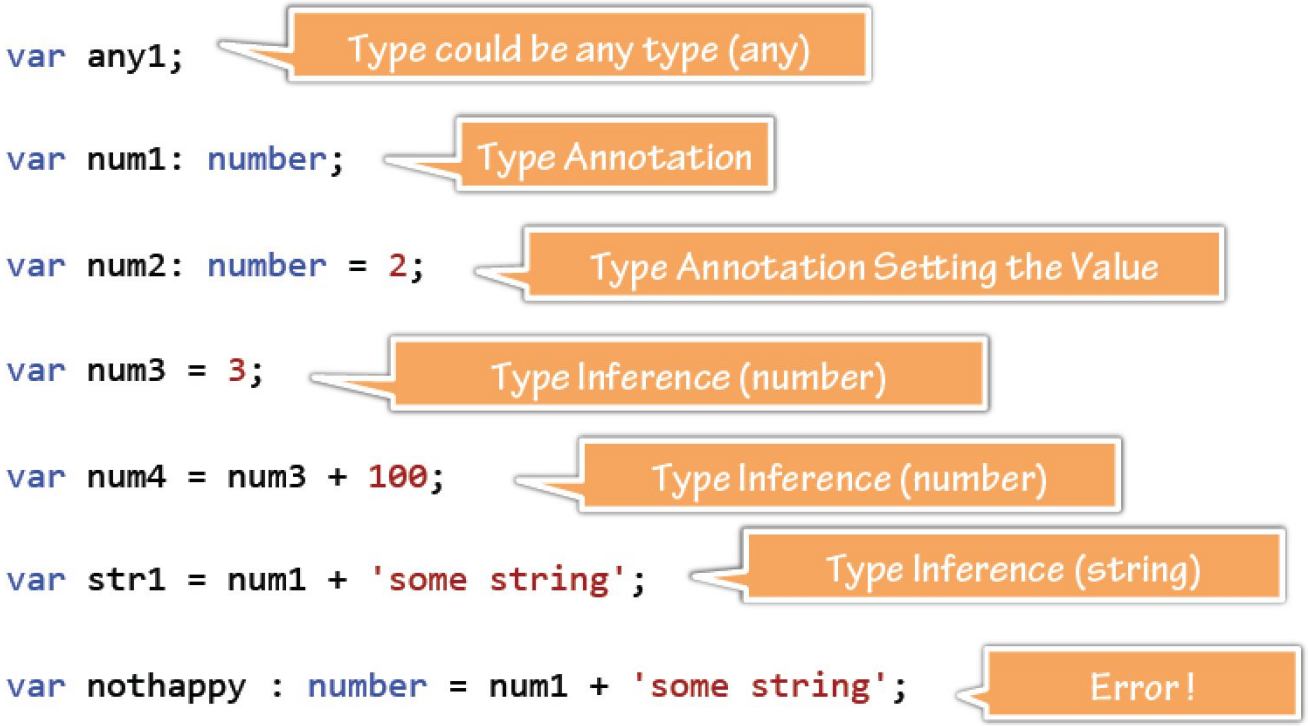
Important Keywords and Operators

Keyword	Description
class	Container for members such as properties and functions
constructor	Provides initialization functionality in a class
exports	Export a member from a module
extends	Extend a class or interface
implements	Implement an interface
imports	Import a module
interface	Defines code contract that can be implemented by types
module / namespace	Container for classes and other code
public/private	Member visibility modifiers
...	Rest parameter syntax
=>	Arrow syntax used with definitions and functions
<typeName>	< > characters use to cast/convert between types
:	Separator between variable/parameter names and types

Code Hierarchy



Annotations and Inferences



Dynamic and Static

TypeScript	JavaScript
Static typing (optional)	Dynamic typing
Type safety is a compile-time feature	Type safety happens at run-time debugging

Type Definition Files (aka Declaration Source Files)

TypeScript

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

declare var $;

var data = "Hello John";

$("div").text(data);
```

Helps provide types for jquery

JavaScript

```
var data = "Hello John";

$("div").text(data);
```

Ambient Declarations do not appear anywhere in the JavaScript

Functions

- **Parameter types (required and optional)**
- **Arrow function expressions**
  - Compact form of function expressions
  - Omit the function keyword
  - Have scope of “this”
- **Void**
  - Used as the return type for functions that return no value

Arrow Function Expressions

TypeScript

```
var myFunc = function (h: number, w: number) {
    return h * w;
};
```

Omit the function keyword

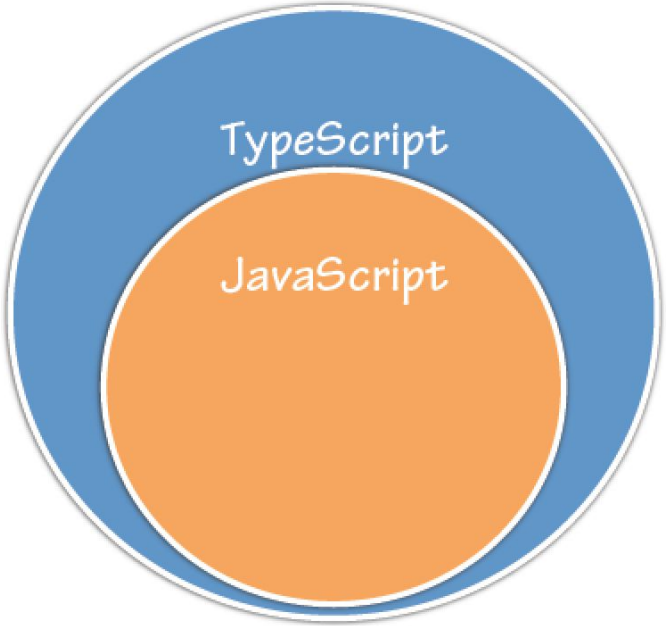
Compact return statement

```
var myFunc = (h: number, w: number) => h * w;
```

Emit the same JavaScript

```
var myFunc = function (h, w) {
    return h * w;
};
```

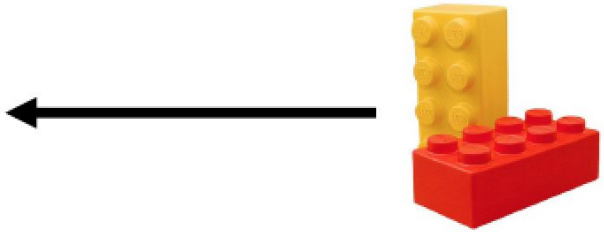
All JavaScript is Valid TypeScript



Typings, Variables and Functions

- **Emits JavaScript**
- **Optional static typing**
  - Various types
- **Compile time checking**
- **Ambient Declarations for external references**
  - Use with typings (*.d.ts files)
- **Objects and functions**
  - Parameter types (required and optional)
  - Arrow function expressions
- **Interfaces**

The Role of Classes in TypeScript



Classes act as containers for different members

TypeScript Class Members

Fields

Constructors

Properties

Functions

Defining a Class

```
class Car {
    //Fields

    //Constructor

    //Properties

    //Functions
}
```

Classes act as containers that encapsulate code

Defining Constructors

Constructors are used to initialize fields

```
class Car {
    engine: string;

    constructor(engine: string) {
        this.engine = engine;
    }
}
```

Field

Constructor

Shorthand way to declare a field

```
class Car {
    constructor(public engine: string) { }
}
```

Adding Functions

```
class Car {
    engine: string;

    constructor (engine: string) {
        this.engine = engine;
    }

    start() {
        return "Started " + this.engine;
    }

    stop() {
        return "Stopped " + this.engine;
    }
}
```

Class members are public by default

Defining Properties

```
class Car {
    private _engine: string;

    constructor(engine: string) {
       this.engine = engine;
    }

    get engine(): string {
        return this._engine;
    }

    set engine(value: string) {
        if (value == undefined) throw 'Supply an Engine!';
        this._engine = value;
    }
}
```

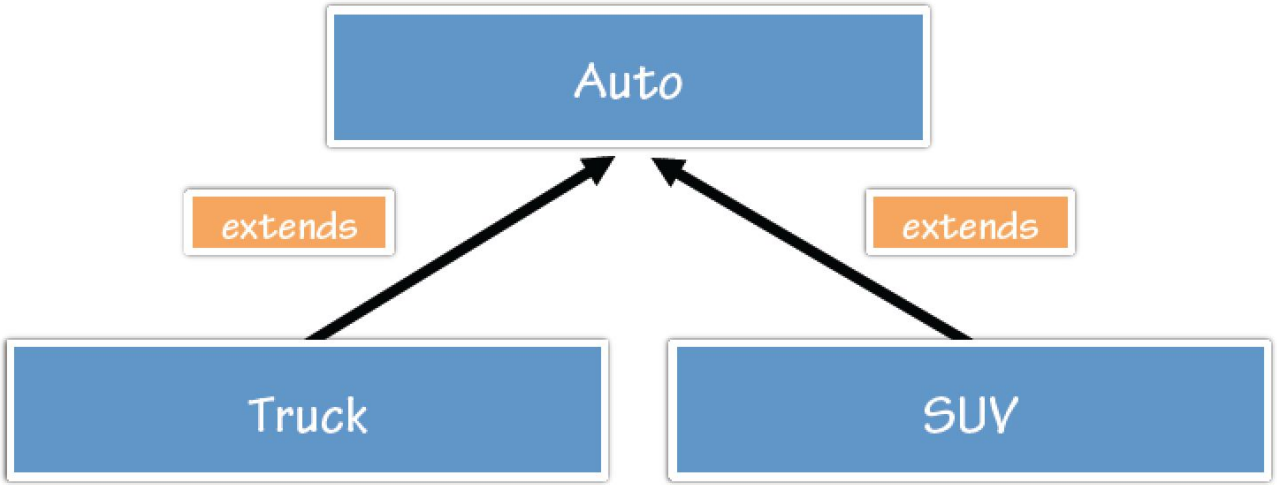
Properties act as filters and can have get or set blocks

Instantiating a Type

Types are instantiated using the "new" keyword

```
var engine = new Engine(300, 'V8');
var car = new Car(engine);
```

Extending Types with TypeScript



Extending a Type

Types can be extended using the TypeScript "extends" keyword

```
class ChildClass extends ParentClass {
    constructor() {
        super();
    }
}
```

Child class constructor must call base class (super) constructor

Type Extension Example

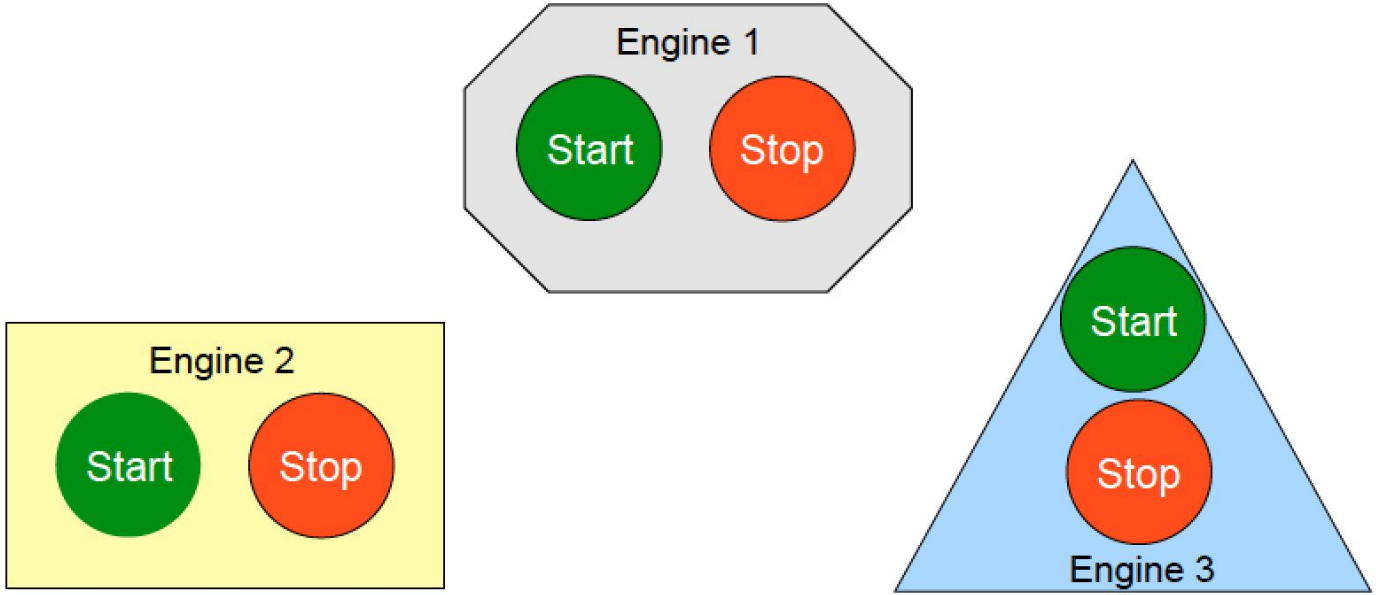
```
class Auto {
    engine: Engine;
    constructor(engine: Engine) {
        this.engine = engine;
    }
}

class Truck extends Auto {
    fourByFour: boolean;
    constructor(engine: Engine, fourByFour: boolean) {
        super(engine);

        this.fourByFour = fourByFour;
    }
}
```

Truck derives from Auto

Call base class constructor

What's an Interface?

- A factory requires that all engines being built have a standard "interface":



Defining an Interface

```
1  interface Action {
2      start(message: string);
3      stop(message: string);
4  }
5
6  class Car implements Action{
7      constructor(public engine: string) {
8          this.engine = engine;
9
10      }
11
12     start(message: string) {
13         console.log(this.engine + message);
14     }
15
16     stop(message: string) {
17         console.log(this.engine + message);
18     }
19
20 }
21
22 var p = new Car('v8');
23
24 p.start('started');
25
```

Interface provide a way to define a “contract” that other objects must implement

- TypeScript provides code encapsulation through classes
- Classes can inherit from other classes
- Interfaces provide a "code contract" to ensure consistency across objects
- Interfaces can extend other interfaces