# A Gentle Introduction to TypeScript *

Bengt-Ove Holländer

bengt.ove.hollaender@gmail.com

* in 45 minutes

# JavaScript

# Origins

In 1995, Brendan Eich got hired at Netscape Communications and faced some challenges:

- 10 days for a working prototype
- Interpretable, embeddable in webpages, lightweight
- Appealing to nonprofessional programmers
- It should look like Java
- But not too much (object-oriented syntax was reserved for Java)
- Advanced Features, but without using new language syntax

In a nutshell *:

- Imperative, dynamic
- Basic syntax from C (e.g. curly braces, semicolons, keywords)
- Very few types (`undefined`, `null`, `boolean`, `number`, `string`, `object`)
- Prototype-based
- Many intrinsic objects (e.g. Function, Array, RegExp)
- Only dynamic type checking (e.g. `null is not a function`)
- Type Coercion

* ECMAScript, 5th Edition

In a nutshell *:

- Imperative, dynamic
- Basic syntax from C (e.g. curly braces, semicolons, keywords)
- Very few types (`undefined`, `null`, `boolean`, `number`, `string`, `object`)
- Prototype-based
- Many intrinsic objects (e.g. Function, Array, RegExp)
- Only dynamic type checking (e.g. `null is not a function`)
- Type Coercion

* ECMAScript, 5th Edition

Stack Overflow Developer Survey 2017:

- 66.7% of professional developers use it
- 59.8% express interest in continuing to develop with it
- Top programming language for web and desktop developers, system administrators and data scientists

Stack Overflow Developer Survey 2017:

- 66.7% of professional developers use it
- 59.8% express interest in continuing to develop with it
- Top programming language for web and desktop developers, system administrators and data scientists

Target audience has changed

Stack Overflow Developer Survey 2017:

- 66.7% of professional developers use it
- 59.8% express interest in continuing to develop with it
- Top programming language for web and desktop developers, system administrators and data scientists

Target audience has changed

Projects are big and business-critical

# Enter TypeScript

# Basics

A richer development toolset for the field of JavaScript programming:

- TypeScript language + compiler
- Language Service that provides completions, code formatting, refactorings etc.

Developed at Microsoft, but open-source since 2012 (0.8.0):

- Good adoption rate, e.g. main language for Google's Angular2

# Basics

TypeScript is a syntactic sugar for JavaScript:

- Syntax is a superset of ES2015 (more recent JavaScript specification)
- Compiler transforms TypeScript programs to human readable JavaScript programs
- Preserves runtime behavior of all JavaScript code
- Introduces no (or minimal) runtime overhead
- Statically identify constructs that are likely to be errors

# Basics

TypeScript is a syntactic sugar for JavaScript:

- Syntax is a superset of ES2015 (more recent JavaScript specification)
- Compiler transforms TypeScript programs to human readable JavaScript programs
- Preserves runtime behavior of all JavaScript code
- Introduces no (or minimal) runtime overhead
- Statically identify constructs that are likely to be errors

# Characteristics

In brief terms:

- Statically type checked - no insertion of runtime checks
- Explicit type annotations
- Type inference
- Structural
- Gradual
- Unsound

# Statically Type Checked

Statically type checked means:

- Types are added to expressions during compilation
- Typing rules and constraints are checked at compile-time
- Violations let the compilation fail
- Programmer can get detailed feedback

## Type System

## Statically Type Checked

## Explicit Type Annotations

```ts
function add2(a: number): number {
    return a + 2;
}

let x: string = 'Hello';

// Argument of type 'string' is not assignable
// to parameter of type 'number'.
add2(x);
```

>>>

## Type System

### Statically Type Checked

### Explicit Type Annotations

### Type Inference

```
function add2(a: number): number {
    return a + 2;
}

let x = 'Hello';

// Argument of type 'string' is not assignable
// to parameter of type 'number'.
add2(x);
```

>>>

Contextual typing

```
type BinaryOp = (a: number, b: number) => number;

function f(g: BinaryOp) {
    return g(2, 3);
}

f(function (a, b) {
    // Type 'string' is not assignable to type 'number'
    a = a.toString();
    return 0;
});
```

>>>

# Structural

In TypeScript, type equivalence and compatibility depends on the type's structure.

```
interface A {
    x: number
}

interface B {
    x: number
}


let a: A = { x: 1 };
let b: B = { x: 2 };

a = b;
```

>>>

Type `B` is compatible with `A` if all members in `A` have a corresponding, compatibly typed member in `B` with the same name.

# Structural

In TypeScript, type equivalence and compatibility depends on the type's structure.

```
interface A {
    x: number
}

interface B {
    x: number
    y: string
}

let a: A = { x: 1 };
let b: B = { x: 2, y: 'Hello' };

a = b;
```

>>>

Type `B` is compatible with `A` if all members in `A` have a corresponding, compatibly typed member in `B` with the same name.

Structural

Gradual

A gradual type system allows parts of a program to be untyped.

- Ensures compatibility of legacy JavaScript
- Idea: Introduce an unknown type (`any` in TypeScript)
- All types can be implicitly converted to `any` and vice versa
- Can not be modeled via subtyping, because it is not transitive

## Type System

### Structural

### Gradual

Expressions without annotations *can* be implicitly inferred to `any`.

- Once a program is fully typed, `--noImplicitAny` can be activated

```
// Parameter 'x' implicitly has an 'any' type
function f(x) {
    return x;
}
```

>>>

## Type System

Structural

Gradual

Unsound

A type system is called sound if a compiler only accepts programs, that never fail with run-time type errors.

- TypeScript is deliberately unsound
- Two examples for unsoundness
  - Covariance of property types
  - Covariance of parameter types

## Type System

Structural

Gradual

**Unsound**

Covariance of property types

```
interface Pet {
    name: string
}

class Cat implements Pet {
    constructor(public name: string) { }
    // is equivalent to
    // constructor(name: string) { this.name = name }

    miaow() {
        console.log('Miaow!');
    }
}

class Dog implements Pet {
    constructor(public name: string) { }

    woof() {
        console.log('Woof!');
    }
}
```

Type System

Structural

Gradual

Unsound

Covariance of property types

```
interface Person {
    pet: Pet
}

class DogLover implements Person {
    constructor(public pet: Dog) { }

    giveTreat() {
        this.pet.woof();
    }
}
```

```
const p = new DogLover(new Dog('Bobby'));

const ref: Person = p;
ref.pet = new Cat('Kitty');

// Uncaught TypeError: this.pet.woof is not a function
p.giveTreat();
```

>>>

Type System

Structural

Gradual

Unsound

Covariance of property types - fix

```
interface Person {
    readonly pet: Pet
}

class DogLover implements Person {
    constructor(readonly pet: Dog) { }

    giveTreat() {
        this.pet.woof();
    }
}
```

```
const p = new DogLover(new Dog('Bobby'));

const ref: Person = p;
// Cannot assign to 'pet' because it is a constant or
// a read-only property.
ref.pet = new Cat('Kitty');
```

>>>

Type System

Structural

Gradual

Unsound

Covariance of parameter types

```
type PetHandler = (Pet) => void

class Family {
    constructor(public pets: Pet[]) { }

    takeCare(petHandler: PetHandler) {
        for (let pet of this.pets) {
            petHandler(pet);
        }
    }
}
```

```
const happyFamily = new Family(
    [new Cat('Kitty'), new Dog('Bobby')]
);

function dogOnlyHandler(dog: Dog) {
    dog.woof();
}

// Uncaught TypeError: dog.woof is not a function
happyFamily.takeCare(dogOnlyHandler);
```

>>>

## Type System

Structural

Gradual

Unsound

Covariance of parameter types - wait, but why?

```
function listenEvent(
    eventType: EventType,
    handler: (n: Event) => void
) {
    /* ... */
}

// Unsound, but useful and common
listenEvent(
    EventType.Mouse,
    (e: MouseEvent) => console.log(e.x + "," + e.y)
);
```

# Advanced Features

# Intersections

Intersection types enable constraints for multiple interfaces.

- Useful for mixins

```
interface A {
    a: number
}

interface B {
    b: number
}

type both = A & B;
```

>>>

# Unions

A union type describes a value that can be one of several types.

```
type either = A | B;
```

>>>

Union types are *very* useful, e.g. when used with `--strictNullChecks`

```
  function xIfGt42(x: number) {
    if (x > 42) {
        return x;
    } else {
        return null;
    }
}

// Type 'number | null' is not assignable to
// type 'number'. Type 'null' is not assignable
// to type 'number'.
const x: number = xIfGt42(0);
```

>>>

A type guard performs a runtime check to guarantee a certain type in a scope.

- Return type is a type predicate `parameterName is Type`

```
function isNotNull(x: number | null): x is number {
    return x !== null;
}

let x = xIfGt42(0);
if (isNotNull(x)) {
    const y: number = x;
}
```

>>>

This custom type guard is actually not required, because it works out-of-the-box.

Type guards also work in switches!

Intersections

Unions

Type Guards

## String Literal Types

String literal types allow the specification of exact values a string must have.

```
function greet(greeting: 'Hello' | 'Hi') {
    console.log(greeting);
}

// Argument of type '"Bye"' is not assignable to
// parameter of type '"Hello" | "Hi"'.
greet('Bye');
```

>>>

# Discriminated Unions

By Example

# Expression Language

Let's implement a language for a very simple form of arithmetic expressions.

- It consists solely of Literals and Additions, e.g. `1` and `1 + 2` are in the language

```
Exp ::= Lit | Add
Lit ::= integer
Add ::= Exp '+' Exp
```

>>>

# Some things I haven't talked about

## Unmentioned

# More TypeScript

Generics

- Similar to those in Java

## Unmentioned

# More TypeScript

Generics

Declaration Merging

- Subsequent interface declarations get merged

>>>

## Unmentioned

# More TypeScript

Generics

Declaration Merging

Polymorphic `this` types

- Useful for fluent interfaces

## Unmentioned

# More TypeScript

Generics

Declaration Merging

Polymorphis `this` types

Index Types

- For checking dynamic property names

>>>

## Unmentioned

# More TypeScript

Generics

Declaration Merging

Polymorphis `this` types

Index Types

Declaration Files

- Contain type specifications for existing JavaScript libraries

Thank you very much!