adesso business. people. technology.

Workshop

JavaScript & TypeScript

JavaScript

ES2015

- arrow functions
- Classes
- Template literals & tagged template literals
- Destructuring
- Rest & Spread
- default parameter, rest parameter
- Modules
- let & const
- Promises
- for..of
- generators
- Map & Set
- Symbols

ES2015+

ES2016

- exponential operator **
- Array.prototype.includes

ES2017

- async / await

ES2018

- Promise.prototype.finally
- RegExp Lookbehind
- Asynchronous Iterators

ES2019

- string trimming
- Array.prototype.[flat, flatMap]

TypeScript

History

- internally developed at Microsoft 2010
- lead by Anders Hejlsberg (lead architect of C#, Turbo Pascal, Delphi)
- public release (0.8) in October 2012
- open source on GitHub
- 1.0 release in 2014
- 2019 Top 3 fastest growing languages on GitHub (https://octoverse.github.com/projects#languages)

What is TypeScript?

- Language Specification
- Typed Superset of EcmaScript
- Compiler
- Dev-Tools
- Transpiler

Why TypeScript?

- Statement completion and code refactoring
- Symbol-based navigation
- Avoids simple tests (expect(service.get).toBeDefined)
- ES2015+ Features
- Less runtime errors (“fail early“)

The result: better maintenance for long-living projects

What is TypeScript not?

- TypeScript has no runtime
- No runtime type checking
- missing functionality has to be polyfilled

Basic types in TypeScript

Types exist for primitive and object types.

```
let isDone: boolean = true;
let size: number = 42;
let firstName: string = 'Lena';
let attendees: string[] = ['Elias', 'Anna'];
```

Tuples are fixed size array

```
let x: [string, number];
x = ["hello", 10]; // OK
x = [10, "hello"]; // Error
```

object

```
let person: {firstName: string, lastName: string}
```

Enums

number based enums

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

String based enums

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```

Functions

function types

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

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

function types

```
type AddFunction = (x: number, y: number) => number

let myAdd: AddFunction = (x, y) => x + y;
```

Optional arguments

Parameter can be optional. Use a question mark.

```
function buildName(firstName: string, lastName?: st
    if (lastName) {
        return firstName + ' '+ lastName;
    } else {
        return firstName;
    }
}
```

Default parameters

Function arguments can have defaults for arguments.

```
function buildName(
    firstName: string,
    lastName: string = 'Adams') {
    // type Inference: lastName is a string
    return firstName + ' ' + lastName;
}
```

Overloads

```
function greet(person: Person)
function greet(name: string)
function greet(personOrName: Person | name) {
    // implementation has to check which
    // arguments are supplied
}
```

Special types

any

- can take any type
- no accessor type checking
- can be used for migrations/external libraries, but should be avoided

```
let question: any = 'Can be a string';
question = 6 * 7;
question = false;
let answer = question.answer; // inferred answer: a
```

Special types - void

value is `null` or `undefined`.

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

Special types - never

Function returning never must have unreachable end point

```
function error(message: string): never {
    throw new Error(message);
}

function forever(): never {
    while(true) { }
}
```

Interfaces

- Type-checking of the shape of values
- Interfaces give a type to these shapes

Interfaces

- Type-checking of the shape of values
- Interfaces give a type to these shapes

Interfaces - Without an interface

You can generate interfaces on the fly.

```
const book: { isbn: string, title: string };

book = {
    isbn: '978-1593272821',
    title: 'Eloquent JavaScript'
};
```

Interfaces - Without an interface

Give an interface a name and use it as a type for variables.

```
interface Book {
    isbn: string;
    title: string;
}
const book: Book;

book = {
    isbn: '978-1593272821',
    title: 'Eloquent JavaScript'
};
```

Interfaces - Optional properties

Properties can be optional.

```
interface Book {
    isbn: string;
    title: string;
    pages?: number;
}
```

Interfaces - Class types

Forgetting to implement ngOnInit throws a compile error.

```
interface OnInit {
    ngOnInit();
}

class BookListComponent implements OnInit {
    ngOnInit() {
    }
}
```

Interface for callable functions

```
interface Greeter {
    (message: string) => void;
}

const g: Greeter = (m: string) => console.log(m);
g('Hello World!')
```

type alias

```
interface Book {
    isbn: string;
    title: string;
}
```

```
type Book = {
    isbn: string;
    title: string;
}
```

type alias are mostly usable like interfaces but can give a name to primitives and tuples and must be used for union and intersection types. These are mentioned in 'Advanced Types'.

A second more important difference is that type aliases cannot be extended or implemented from (nor can they extend/implement other types). Because an ideal property of software is being open to extension, you should always use an interface over a type alias if possible.

Classes

```
class Person {
    readonly name: string;
    firstName: string; // public by default
    protected age: number;
    private salary: number = 0;

    constructor(name: string, age: number, sala
        this.name = name; this.age = age; this.
    }

    sayHi() {
        this.toConsole("Hi, I'm"):
    }
```

Class generates multiple declarations

- Contructor function
- instance type

```
let p: Person; // instance type
p = new Person(); // constructor function

// type of constructor function
let PersonCtr: typeof Person;
PersonCtr = Person;
p = new PersonCtr();
```

instance type of class can be used as interface

```
class Point {
    x: number;
    y: number;
}

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

Generics

Generics can be used in

- interfaces
- type aliases
- classes
- methods
- functions.

class with generic

```
class Box<T> {
    data: T;
    constructor(data: T) {
        this.data = data
    }
}

const stringBox = new Box('Hello World!');
const msg: string = stringBox.data;
```

function with generic

```
function identity<T>(arg: T): T {
    return arg;
}

// infered type of output: string
let output = identity("myString");
let output = identity<string>("myString");
```

Generic Constraints

```
function personLogger<T extends Person>(p: T) {
    console.log(p.name);
}

personLogger(new Person()) // OK
personLogger('John Doe') // Error
```

Advanced Types

Union types

model the possible different types of parameters, that are typical for dynamic languages.

```
print(toPrint: string | string[]): void {
    if (typeof toPrint === 'string')
        console.log(toPrint);
    else
        console.log(toPrint.join('\n'))
}
```

Intersection types

combines types without direct inheritance (mixins/composition).

```
interface xDimension {x: number}
interface yDimension {y: number}
interface zDimension {z: number}

type Point3d = xDimension & yDimension & zDimension
// interface Point3d extends xDimension, yDimension

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

Intersection types - recursive structures

```
type LinkedList<T> = T & { next: LinkedList<T> };
interface Person {
    name: string;
}
var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
```

String/numeric literal types

constant string or numeric values can be also be a type

```
let five: 5 = 5
five = 6; // Error
```

```
let foo: 'foo' = 'foo';
foo = 'bar'; // Error
```

String/numeric literal types

useful in combination with union types

```
function log(level: 'info' | 'warn' | 'error', msg:
    // ...
}

log('info', 'Hello World!')
log('debug', 'not possible') // Error
```

Index Types

allow type checked access for properties on objects.

```
interface Person {
    name: string;
    age: number;
}

let personKeys: keyof Person; // 'name' | 'age'
```

```
// extract a property from an object
function pluck<T, K extends keyof T>(o: T, key:
  return o[n];
}
let john: Person = {
    name: 'John',
    age: 35
};
const age = pluck(john, 'age') // type: number
```

index types in interfaces

```ts
interface Dictionary<T> {
    [key: string]: T;
}

const foo: Dictionary<number> = {};
foo['first'] = 1; // OK
foo['second'] = 'hello' // Error;
```

Mapped Types

allow generic transformations for each property of an old type to a new type

build in mapped types

```
type Partial<T> = { [P in keyof T]?: T[P]; };
type Pick<T, K extends keyof T> = { [P in K]: T[P];
type Readonly<T> = { readonly [P in keyof T]: T[P];
type Required<T> = { [P in keyof T]-?: T[P]; };
```

Conditional Types

```
type Exclude<T, U> = T extends U ? never : T;
type Extract<T, U> = T extends U ? T : never;
type Parameters<T extends (...args: any) => any> =
    T extends (...args: infer P) => any ? P : never
type ConstructorParameters<T extends new (...args:
    T extends new (...args: infer P) => any ? P : n
type ReturnType<T extends (...args: any) => any> =
    T extends (...args: any) => infer R ? R : any;
type InstanceType<T extends new (...args: any) => a
    T extends new (...args: any) => infer R ? R : a
```

Type compatibility

- structurally typed
- same shapes are compatible
- type is only a name for the shape

structural typing

```
interface Named {
    name: string;
}
class Person {
    name: string;
}
// OK, because of structural typing
const p: Named =  new Person();
```

structural typing

```
class Person {
    name: string;
    lastName: string;
}

// also OK, because additional properties are not c
const p: Named = new Person();
```

Type guards

- narrow down union types
- check inputs at application borders (user input, API calls)

Type guards - build in type guards

- instanceof type guards
- typeof type guard

```
function greet(maybePersonOrName?: Person | name) {
    if(maybePersonOrName instanceof Person)){
        console.log('Hi', maybePersonOrName.name);
    } else if (typeof maybePersonOrName === 'string
        console.log('Hi', maybePersonOrName);
    } else { // maybePersonOrName === undefined
        console.log('Hi!')
    }
}
```

Type guards - custom type guards

- function returning a boolean value to signal the type of the parameter.
- return type of a custom type guard is a special form of `paramName is Foo`.

```
function isFish(pet: Fish | Bird): pet is Fish {
    return (<Fish>pet).swim !== undefined;
}
```

Decorators

- experimental feature (may change in future versions)
- heavily used in Angular
- allow AOP

must be enabled in tsconfig.json

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

Decorators

- decorators for classes, methods, properties and method parameters
- docorators are simple functions, that are called with

How to decorate in ES5

Decorators, or higher order functions for classes in ES5 are simple

```
function Robot(target) {
  target.isRobot = true;
}

function Number5() {…}

Robot(Number5);

Number5.isRobot; // ==> true
```

How to decorate a ES2015/TS class

The constructor function can be notated as class

```
function Robot(target) {  target.isRobot = true;}

class Number5 {…}  Robot(Number5);

Number5.isRobot; // ==> true
```

But the isRobot call belongs directly to Number5

How to decorate a ES2016/TS

Since the decorator function is just a function, it can be a Higher Order Function to get configuration parameters.

```
function Robot(roboName) {
    return function(target) {
        target.name = roboName;
    }
}

@Robot('Bender')
class Number5 {…}  Number5.name; // ==>'Bender'
```

Decorators

real world usage: Angular Component

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @Input()
  title = 'Tour of Heroes';
}
```

Decorators

real world usage: nest.js, a Node.js web framework

```
@Controller('cats')
export class CatsController {
  @Get()
  findAll(@Req() request: Request): string {
    return 'This action returns all cats';
  }

  @Post()
  @HttpCode(204)
  create() {
```

JavaScript interop

TypeScript is a superset of JavaScript, so every library written in JavaScript must be usable in TypeScript.

load untyped JavaScript Modules

Declaration Files

- Declaration files are simmilar to C/C++ header files.
- No runnable code, only type information for TypeScript compiler
- Can be written for external libraries

custom declarations

lodash.d.ts

```
declare module 'lodash' {
    isArray(o: any): boolean;
    keys<T>(o: T): keyof T[];
    //...
}
```

npm @types

Declaration files for well known packages are published in the npm registry under the @types namespace. Examples are `@types/node` for Node.js, `@types/lodash`, ...

Best practices

- Always type parameter and return value

Hands-on

```
mkdir hands-on
cd hands-on
npm init -y
npm i typescript
npx tsc --init
```