JAVASCRIPT

- The world's best language
- And the world's worst language

BEST

The demand for web developers is higher than ever. "Front-end programming" becomes more important with each year, as more development happens on the front end. With Node.js, JavaScript is making in-roads on the back end.

WORST

Poor debugging, obscure error messages, frustrating platform discrepancies. No static types. Confusing implicit type coercion. Traditionally — fixed in ES6 — no classes, no modules.

TypeScript is a superset on top of Javascript. It offers us two important things:

- Most importantly, it adds static types to JavaScript.
- While we're adding a pre-compilation step anyway, why not allow the newest features from ES6?

WHY DO WE NEED TO IMPROVE JAVASCRIPT?

Let's look at some common JavaScript problems. Adding numbers makes sense, as does "adding" (concatenating) strings:

```
let n = 10;
 n + 32;                // 42
 let name = "Chthulu";
 "Hello, " + name;  // "Hello, Chthulu!"
```

But what happens when we have mixed types?

```
let customer = {
     name: "Jones",
     orders: 172
 };
 1 + customer;       // "1[object Object]" (!?)
```

JavaScript does not restrict how functions are called: you can call a function with too may arguments, too few, or wrong order or wrong types:

```
function repeat(s, n) {
     let result = "";
     for (let i = 0; i < n; i++) {
         result += s;
    }
    return result;
}
repeat("w", 3);
repeat("w", 3, "um");
repeat();
repeat(3, "w");
//  "www"
//  "www"  (too many args)
//  ""     (too few)
//  ""     (wrong order/type)
```

In our previous example, we could compensate by explicitly checking parameters. Even then, the function fails at run time. We can do better. Some more pitfalls:

- Mistyping properties and methods
- Passing the wrong types of arguments
- Using return values wrongly

Best of all: it's still JavaScript

Many other compile-to-JS languages (CoffeeScript, Dart, Elm) replace JavaScript's syntax by a new one. TypeScript, on the other hand, only extends the syntax — everything you know about JavaScript-the-language is still true when you code TypeScript.

TYPESCRIPT SYNTAX

THE BASIC TYPES

Contrary to popular belief, JavaScript does have types.
It has exactly six primary types:

```
let active = true;
let name = "Eleonore";
let legs = 4;
let obj = { n: 42 };
let noObj = null;
let u;
// boolean
// string
// number
// object (including functions and arrays)
// null
// undefined
```

So the first thing we can do to exploit explicit typing is to declare the type of a variable:

```
    let person: string = "Sandra";
```

Though perhaps a more realistic example is this:

```
let person: string = getPerson();
```

Suddenly we're declaring expectations on the type of a return value. That's useful.

If you assign a constant value without declaring the type, the type will be declared for you. TypeScript figures it out.

```
    let person = "Petra";    // variable person is now typed '
    person = 3.14;           // ERROR: Type 'number' is not as
                             //        to type 'string'
```

In JavaScript, nothing prevents you from stuffing a number into a variable that used to hold a string. In TypeScript, the default is to disallow that.

So the default is "no funny business". What if, for whatever reason, you want to do weird things to a variable that you know the compiler won't like?

Answer: use the any type. It opts you out of the type checker.

```
let person: any = "Petra";
person = 3.14;              // suddenly fine
console.log(person.foo);     // also fine
```

It goes without saying that any should not be overused. You wanted TypeScript, but now you're turning it off? In fact, any switches you back down to good ol' unsafe JavaScript.

When we declare a function, we have the option to declare a type both for each parameter in the function, and for the function's return value:

```
 function repeat(s: string, n: number): string {
     return new Array(n + 1).join(s);
}
```

Parameters get typed as any unless you explicitly type them.

There's a type called void that really only makes sense for return types. It carries the semantics of "this function is not supposed to return anything".

```
    function cleanSideEffect(): void {
        // you're allowed to do `return;` in here
        // or `return null;` or `return undefined;`
        // but not return any other values
}
```

INTERFACES

Typescript gives ut the possibility to define interfaces

```
// The position of a chess square, such as 'e6'
interface ChessPosition {
    file: string;
    rank: number;
}
let pos: ChessPosition = { rank: 6, file: "e" };
```

TypeScript's form of typechecking is called structural (as opposed to the mainstream nominal typechecking).

- Structural: it's a ChessPosition because it has the right parts
- Nominal: it's a ChessPosition because it's an instance of the type with that name

An interface can also contain optional properties:

```
interface Employee {
     name: string;
     salary?: number;
     superior?: Employee;
}
```

CLASSES

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach.

Typescript lets us create classes, that then compiles down to prototype-based inheritence.

We construct classes using the class syntax.

```
class User {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    sayHi() {
        return "Hello, " + this.name;
    }
```

The class takes a method **constructor** that runs when a new instance is created.

```
class User {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
}
const user = new User() // constructor is executed
```

We can inherit classes using **extends**

```
class SpecificUser extends User {
}
```

We can use **abstract classes** to create interfaces. Abstract classes can't be instantiated by itself, but can be inherited.

```
abstract class User {
    sayHi() {}
}
class SpecificUser extends User {
}
```

We can use **Public**, **private**, and **protected** modifiers.

```
class User {
    public name: string; // public to all
    private id: number; // closed to all except internal
    protected meta: number; // open only to those who inherit
}
```

By default, method and properties are **public**

We can use **static properties** to mark methods accessable outside instances.

```
class User {
    static settings: string;
}
User.settings // available outside instance
```

MODULES

Starting with ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

We can import modules using **import**

```
import { loadData } from "./helpers";
```

We can export modules using **export**

```
export function loadData() {
}
```