TypeScript

Компилятор

Код на TypeScript компилируется в JS и подходит для разработки любых проектов под любые браузеры — тем более что можно выбрать версию JS, в которую будет компилироваться код.

Основная задачи TypeScript принести типы в js.

```
#глобальная установка ts
npm i -g typescript
#конфигурация, которая описывает как мы компилируем в js
tsc --init
#компилируем
tsc
```

Interfaces and Types

У Interfaces и Types есть три главных различия

1. Нельзя использовать implements для types, если присутствует оператор union.

   ```
   type Box = Shape | Figure
   class MyBox implements Box {} // ошибка
   ```

2. Нельзя использовать одно имя для нескольких типов. Другими словами, нельзя расширить тип.

   ```
   //Свойства интерфейсов можно расширять
   interface Test {
    a: number
   }
   interface Test {
    b: number
   }
   ```

3. Нельзя наследовать тип от типа.

   ```
   interface Test{
    a:number
   }
   interface Test2 extends Test{
    b:string
   }
   // Теперь Test2 созержит и свойство a и свойство b
   ```

Рекомендуется использовать именно interfaces, а не types

Types удобно использовать для создания своего типа, с использованием union оператора

```
type stringOrNumber = string | number
```

Литеральные типы

Литеральные типы полезны, когда мы хотим ограничить область значения переменной

```
type CardinalDirection = 'Up' | 'Down';
function move(distance: number, direction: CardinalDirection): -1 | 1 | 0 {
    switch (direction) {
        case "Up":
            return -1
        case "Down":
            return 1
        default:
            return 0
    }
}
```

Enums

Зачем нужен Enums?

Константа в ООП доступна только в пределах класса. Что если я хочу хранить список цветов и использовать их при отрисовке фигур во всем моем проекте? Можем создать отдельный класс для списка цветов, только чтобы хранить там варианты, и затем использовать параметры этого типа во всех методах отрисовки фигур - уже неплохо, мы добилить типизации, так?... но нам не нужен ЦЕЛЫЙ класс для этого...никакое состояние мы там не храним, поведение не определяем...для этого и придумали enum.

```
enum Colors {
  White,
  Black
}

class Rectangle {
  DrawUsingEnum(color: Colors){
    switch(color){
      case Colors.White: {
        //..draw white rect
      }
      case Colors.Black: {
        //..draw black rect
      }
    }
  }

  DrawUsingNumber(color: number){
    switch(color){
      case 1: { //let say you draw the WHITE one when you passed 1
```

```
        //..draw white
      }
      case 2: { //it's for black
        //..draw BLACK one
      }
    }
  }

  DrawUsingString(color: string) {
    switch(color){
      case "white": {
        //...draw white rect
      }
      case "black": {
        //...draw black rect
      }
    }
  }
}

class Program{
  Main() {
    var rect1 = new Rectangle();
    rect1.DrawUsingEnum(Colors.White) // pass enum as a parameter. No WAY to pass
smth else like Colors.Red

    rect1.DrawUsingNumber(999) // what happens here??? we handle only 1 & 2
    rect1.DrawUsingString("red") // again we do not have case for "red"
  }
}
```

Tuple

Кортеж (tuple) упорядоченный набор фиксированной длины. Кортеж, как структура данных, примечателен тем, что в него нельзя добавлять элементы, а так же нельзя менять местами элементы и нарушать порядок следования.

```
const a: [number, string, number] = [1, "2", 3]
```

Generics

Дженерики — это возможность создавать компоненты, работающие не только с одним, а с несколькими типами данных

Дженерики и типы соотносятся друг с другом, как значения и аргументы функции. Это такой способ сообщить компонентам (функциям, классам или интерфейсам), какой тип необходимо использовать при их вызове так же, как во время вызова мы сообщаем функции, какие значения использовать в качестве аргументов.

```
function identity <T>(value: T) : T {
    return value;
}
console.log(identity<Number>(1))
console.log(identity<String>("test"))
```

JSX

Расширение файлов: jsx, tsx.

В конфиге:

```
{
  "jsx": "react"
}
```

Устанавливаем библиотеку, которая даст нам типизацию

```
npm i react
npm i -D @types/react
```

Аналогичное создание компонентов

```
const a: JSX.Element = <div tabIndex={0}> Test </div>;
const b: JSX.Element = React.createElement("div", {tabIndex:1}, 'Test');

// компилируется в
"use strict";
var __importDefault = (this && this.__importDefault) || function (mod) {
    return (mod && mod.__esModule) ? mod : { "default": mod };
};
exports.__esModule = true;
var react_1 = __importDefault(require("react"));
var a = react_1["default"].createElement("div", { tabIndex: 0 }, " Test ");
var b = react_1["default"].createElement("div", { tabIndex: 1 }, 'Test');
```