нетология

TYPESCRIPT

ВЛАДИМИР ЧЕБУКИН

ВЛАДИМИР ЧЕБУКИН

Frontend-разработчик

vovachebr@mail.ru fb.me/vovachebr @User123423

ПЛАН ЗАНЯТИЯ

1. TypeScript
2. Рабочее окружение
3. Типизация
4. Интерфейсы
5. Классы
6. TypeScript & Webpack

TYPESCRIPT

TYPESCRIPT

У нас сегодня достаточно обширная лекция, в которой мы:

- рассмотрим основы TypeScript
- напишем на нём небольшое приложение
- а также встроим TypeScript в наш шаблон Webpack

TYPESCRIPT

TypeScript - это язык, в первую очередь, добавляющий в JS возможности типизации (а именно возможность объявления типов и проверки их использования на этапе компиляции).

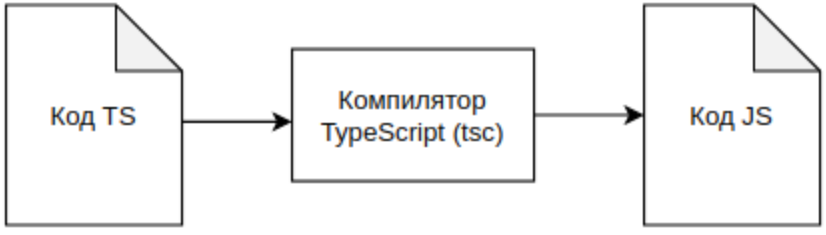
Q: Что это значит?

A: Это значит, что в отличие от JS мы можем указывать для переменных, полей и параметров типы, а сам компилятор будет следить за правильностью их использования (что вы не кладёте в число строку и т.д.) при компиляции.

Q: Компиляция?

КАК РАБОТАЕТ TYPESCRIPT

Компилятор TypeScript преобразует код на языке TypeScript в код на языке JavaScript (необходимой версии).



Q: Очень похоже на Babel

A: Похоже, но не совсем, т.к. всё-таки TypeScript - это другой язык программирования

ЗАЧЕМ ЗНАТЬ TYPESCRIPT?

Q: TypeScript нужен только для проверки типов на этапе компиляции?

A: Нет, на самом деле TypeScript поддерживает достаточно много возможностей, которые появятся только в будущих версиях JS (либо не появятся вообще).

Кроме того, TypeScript сейчас используется повсеместно, особенно в больших проектах, т.к. позволяет отловить множество мелких ошибок на этапе компиляции.

ЗАЧЕМ ЗНАТЬ TYPESCRIPT?

44% JavaScript-разработчиков регулярно пишут на TypeScript. В общей сложности в 2019 году этот язык использует четверть всех разработчиков, по сравнению с 17% в прошлом году.

Исследование JetBrains за 2019 год.

РАБОЧЕЕ ОКРУЖЕНИЕ

РАБОЧЕЕ ОКРУЖЕНИЕ

Просто так использовать TypeScript не получится, т.к. браузеры его не понимают.

Мы, конечно, можем поэксперементировать в Playground - интерактивной песочнице, но так не интересно.

Поэтому нам необходимо настроить наше рабочее окружение - сначала мы это сделаем без Webpack, а затем уже и с Webpack.

НАСТРОЙКА ПРОЕКТА

```
npm init
```

Установим TypeScript как dev-зависимость:

```
npm install --save-dev typescript
```

Создадим файл конфигурации компилятора TypeScript:

```
npx tsc --init
```

TSCONFIG.JSON

Файл конфигурации достаточно большой (и опции в нём задокументированы), но нас будут интересовать всего несколько:

- `target` - целевая версия ES (по умолчанию выставляется ES5)
- `module` - целевая система модулей (по умолчанию выставляется CommonJS)
- `outFile` - при необходимости скомпилировать всё в единый файл
- `outDir` - каталог для результатов компиляции
- `rootDir` - каталог с исходными файлами
- `sourceMap` - генерация Source Map (возможности отображать скомпилированный код в исходный при работе в дебаггере)

TSCONFIG.JSON

Файл `tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "sourceMap": true,
    "outDir": "./dist",
    "rootDir": "./src",
    ...
  }
}
```

Файл `package.json`:

```
{
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w"
  },
}
```

WATCH MODE

Запускает компилятор в специальном режиме "отслеживания изменений и перекомпиляции":

`npm run watch`

Наберём следующий код в `src/demo.ts`:

```
const message = 'hello world';
console.log('message')
```

В результате компиляции получим в `dist/demo.js`:

```
"use strict";
var message = 'hello world';
console.log('message');
//# sourceMappingURL=demo.js.map
```

Пока видим только перевод в ES5. Давайте смотреть глубже.

ЗАДАЧА

Мы проектируем достаточно большой веб-портал и хотим организовать на нём функцию Корзины, в которую можно складывать товары, вычислять сумму и бонус (в размере 1% от всей суммы).

Посмотрим, как это сделать с помощью TypeScript.

ТИПИЗАЦИЯ

ТИПИЗАЦИЯ

Первое и самое главное, мы можем указывать типы данных для переменных, параметров и полей (объектов, классов и т.д.):

```
let totalAmount: number = 0;
// ошибка totalAmount = 'много';
```

```
src/index.ts:2:1 - error TS2322: Type '"много"' is not assignable to type 'number'.

2 totalAmount = 'много';
  ~~~~~~~~~~~

[5:04:25 PM] Found 1 error. Watching for file changes.
```

Важно: `let` мы используем только для демонстрации проверки типов при назначении значений (вы должны использовать в своих приложениях по-максимуму `const`).

БАЗОВЫЕ ТИПЫ

Базовые типы соответствуют типам JS:

- `number`
- `boolean`
- `string`

Особняком стоят `null` и `undefined`, которые являются подтипами любого типа*.

Примечание*: но компилятор не генерирует ошибки при таком присваивании только при выключенном "strict"-режиме (а именно `strictNullChecks`).

UNION

TypeScript позволяет указывать допустимые типы для переменной (поля или аргумента), если это необходимо:

```
let errorCode: number | null = null;
errorCode = 404; // ok
```

Символ `|` разделяет допустимые типы.

Нужно достаточно аккуратно работать с этой возможностью не для `null` или `undefined`.

`any`

Тип `any` фактически отключает проверку типов, "возвращая" нас в режим JS, где тип переменной определяется только значением, которое в ней хранится.

Если в вашем коде много `any` - это яркий маркер того, что вы "плохо" пишете на TypeScript.

ДРУГИЕ ТИПЫ

Кроме того, типами могут являться объекты, классы и другие (специфичные для TypeScript) типы вроде `enum`, `void`, `never`.

Мы не будем рассматривать все типы (о них прекрасно написано в документации), рассмотрим лишь самые используемые.

ОБЪЯВЛЕНИЕ ТИПА

Итак мы уже посмотрели с вами, что можем указать тип в определении (или объявлении переменной):

```
let bonus: number;
let totalAmount: number = 0;
```

А для функций мы можем указать как типы параметров, так и тип возвращаемого значения:

```
// для функций
function calculateBonus(amount: number): number {
  return Math.ceil(amount * 0.01);
}
```

Если функция ничего не возвращает и мы явно это хотим указать, то для этого есть специальный тип `void`.

ВЫВЕДЕНИЕ ТИПОВ

Компилятор TypeScript достаточно умный, и там, где есть явная инициализация, сам может вывести тип:

```
// было:
let totalAmount: number = 0;
// стало:
let totalAmount = 0;
// number будет выведено автоматически
```

Конечно же, это не будет работать для параметров функций.

СЛОЖНЫЕ ТИПЫ

Хорошо, для примитивов и `any` вроде как понятно, но какой тип будет для такого объекта*?

```
let item = {
  id: 1008,
  name: 'Meteora',
  author: 'Linkin Park',
};
```

И самый главный вопрос: как прописать этот тип для параметра функции:

```
function addToCard(item): void {
  // TODO:
}
```

Напоминаем, что в JS в цепочке прототипов сразу будет `Object`.

СТРУКТУРНАЯ ТИПИЗАЦИЯ

В TypeScript используется подход структурной типизации: тип объекта определяется его "формой" - набором полей и их типом.

Т.е. происходит примерно следующее:

```
type Item = {
  id: number,
  name: string,
  author: string,
};

let item : Item = {
  id: 1008,
  name: 'Meteora',
  author: 'Linkin Park',
};
```

Но тип `Item` явно не объявляется а просто выводится из структуры текущего объекта.

СТРУКТУРНАЯ ТИПИЗАЦИЯ

Мы можем назначать переменной (полю или параметру) любой объект, который по типу соответствует (содержит ровно такой же набор полей такого же типа):

```
item = {
  id: 1001,
  name: 'War and Piece',
  author: 'Leo Tolstoy',
};
```

СТРУКТУРНАЯ ТИПИЗАЦИЯ

Но попытка "положить" туда объект с доп.полями к успеху не приведёт:

```
23  item = {
24      id: 1001,
25      name: 'War and Piece',
26      author: 'Leo Tolstoy',
27      pages: 1225
28  };
```

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL                    1: npm

src/index.ts:27:5 - error TS2322: Type '{ id: number; name: string; author: string; pages: number; }
' is not assignable to type 'Item'.
  Object literal may only specify known properties, and 'pages' does not exist in type 'Item'.

27      pages: 1225
```

ОТСТУПЛЕНИЕ: OPTIONAL

На самом деле, мы, конечно, можем использовать возможность, которая называется `optional`, создать определение типа и прописать, что в объекте может быть поле `pages`, а может и не быть:

```
type Item = {
  id: number,
  name: string,
  author: string,
  pages?: number,
};
```

Но если на портале продаётся большое количество товаров с разными характеристиками, то объявление типа будет просто огромным.

А нам, чтобы добавить товар в корзину, достаточно знать лишь id товара, его название (автор, в целом, и не нужен) и стоимость.

ИНТЕРФЕЙСЫ

ИНТЕРФЕЙСЫ

TypeScript поддерживает концепцию интерфейсов: в терминах TypeScript требование на наличие определённых свойств (с ограничением на типы) у объекта:

```
interface Buyable {
  id: number,
  name: string,
  price: number,
};

...

function addToCard(item: Buyable): void {
  // TODO:
}
```

Теперь мы можем передавать в эту функцию любые объекты, у которых есть эти три свойства (с нужными типами). При этом наличие/отсутствие остальных свойств нас не интересует.

ИНТЕРФЕЙСЫ

Интерфейсы - одна из ключевых возможностей TypeScript, позволяющая на полную мощность использовать контроль типов.

Доп.возможности интерфейсов:

- optional property - `?` работает так же, как в объявлении типа
- readonly property - `readonly` значение свойства доступно только для чтения (после создания объекта)
- function property - определения типа свойств-функций
- и многое другое

`Buyable`

Поскольку мы будем использовать сборщики (либо можно заставить компилятор TypeScript сделать это самостоятельно), принято разносить типы на разные файлы и не писать всё в одном.

Вынесем наш интерфейс в отдельный файл `domain/Buyable.ts` и экспортируем его по умолчанию (аналогично ESM):

```
export default interface Buyable {
  readonly id: number,
  readonly name: string,
  readonly price: number,
}
```

Осталось лишь определить продукты, которые доступны к продаже и саму корзину.

КЛАССЫ

КЛАССЫ

Классы в TypeScript, в целом, схожи с классами в JS (включая те возможности, которые поддерживает Babel), за исключением ряда концепций:

1. Возможность имплементации интерфейсов
2. Модификаторы доступа для полей и сокращённые форматы инициализации

Создадим по классу для книг и музыкальных альбомов (каталог `domain` файлы `Book.ts`, `MusicAlbum.ts` - не приведён для краткости):

```
import Buyable from './Buyable';

export default class Book implements Buyable {
  readonly id: number;
  readonly name: string;
  readonly author: string;
  readonly price: number;
  readonly pages: number;

  constructor(id: number, name: string, author: string, price: number, pages: number) {
      this.id = id;
      this.name = name;
      this.author = author;
      this.price = price;
      this.pages = pages;
  }
}
```

Имплементируя интерфейс мы обязаны иметь в классе те же самые поля тех же самых типов, что были объявлены в интерфейсе.

МОДИФИКАТОРЫ ДОСТУПА

В TypeScript (скоро будет и в JS, но в другой форме) есть концепция модификаторов доступа - мы можем ограничить использование полей объекта снаружи:

- `public` - доступно всем (по умолчанию, обычно не указывается)
- `protected` - доступно только в наследниках
- `private` - доступно только внутри класса

Компилятор будет информировать нас о попытках неправильного использования полей (например, при доступе к `private` полям).

МОДИФИКАТОРЫ ДОСТУПА

```
import Buyable from './domain/Buyable';

export default class Cart {
    private items: Buyable[] = [];

    add(item: Buyable): void {
        this.items.push(item);
    }

    getAll(): Buyable[] {
        return [...this.items];
    }
}
```

МОДИФИКАТОРЫ ДОСТУПА

```
1   import Cart from './Cart';
2
3   const cart = new Cart();
4   console.log(cart.items);
5
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
[11:18:00 AM] File change detected. Starting incremental compilation...

src/app/index.ts:4:18 - error TS2341: Property 'items' is private and only accessible within class 'Cart'.

4 console.log(cart.items);
                   ~~~~~
```

GET/SET & ETC

get/set и другие возможности JS так же поддерживаются, поэтому предыдущий пример можно было бы переписать вот так:

```
import Buyable from './domain/Buyable';

export default class Cart {
    private _items: Buyable[] = [];

    add(item: Buyable): void {
        this._items.push(item);
    }

    get items(): Buyable[] {
        return [...this._items];
    }
}
```

PARAMETER PROPERTIES

Вернёмся к нашим книгам: TypeScript позволяет использовать сокращённую запись для инициализации свойств (благодаря модификаторам доступа или `readonly`):

```
export default class Book implements Buyable {
    constructor(
        readonly id: number,
        readonly name: string,
        readonly author: string,
        readonly price: number,
        readonly pages: number,
    ) { }
}
```

Таким образом, объявление классов, содержащих только данные (Data Classes) становится максимально удобным.

Вместо `readonly` можно использовать `public`, `protected` или `private` (в зависимости от логики класса).

НАСЛЕДОВАНИЕ

TypeScript поддерживает наследование и мы могли бы организовать работу корзины и без интерфейсов, создав базовый класс `Item` с нужными полями и отнаследовавшись от него (классы `Book` и `MusicAlbum`).

Но в современном мире использование интерфейсов считается более предпочтительным, поскольку наследование является более тесной связью (наследоваться можно только от одного класса, а реализовывать можно сколько угодно интерфейсов).

TYPESCRIPT & WEBPACK

WEBPACK

Просто так смотреть на скомпилированный код не особо интересно, поэтому мы с вами интегрируем написанное нами приложение в наш же шаблон Webpack.

Для этого нам понадобиться всего лишь установить соответствующий loader и сам компилятор typescript:

```
npm install --save-dev typescript ts-loader
```

webpack.config.js

Затем необходимо положить файл конфигурации TypeScript `tsconfig.json` в корень проекта и внести изменения в конфигурационный файл Webpack:

```
module: {
    rules: [
        {
            test: /\.tsx?$/,
            exclude: /node_modules/,
            use: {
                loader: 'ts-loader',
            }
        },
        ...
    ]
},
resolve: {
    extensions: [ '.tsx', '.ts', '.js' ],
},
...
```

JEST

Для поддержки тестирования TypeScript-кода с помощью Jest есть две опции:

1. С использованием Babel (Babel будет транспилировать код TypeScript)
2. ts-jest - специальный инструмент (является наиболее оптимальным решением на сегодняшний день)

Рассмотрим второй вариант:

```
npm install --save-dev ts-jest @types/jest
```

```
npx ts-jest config:init
```

JEST

```
import Cart from '../service/Cart';

test('new card should be empty', () => {
    const cart = new Cart();

    expect(cart.items.length).toBe(0);
});
```

`npx test`

ИТОГИ

ИТОГИ

TypeScript достаточно популярный язык и многие большие проекты пишутся именно на нём, а не на чистом JS.

Поэтому навыки работы с ним будут хорошим инструментом в вашем арсенале.

Мы рассмотрели лишь часть возможностей TypeScript, не касаясь таких тем как Generics (обобщённое программирование), Enum, Decoratos и т.д.

Рекомендуем вам самостоятельно ознакомиться с Handbook'ом, в котором эти возможности описаны достаточно просто и лаконично.

Кроме того, мы не оставили за рамками лекции интеграцию TypeScript с другими инструментами нашей инфраструктуры, такими как ESLint и т.д.

нетология

Задавайте вопросы и напишите отзыв о лекции!

ВЛАДИМИР ЧЕБУКИН

vovachebr@mail.ru

fb.me/vovachebr

@User123423