УДК 004

Преимущества использования TypeScript вместо JavaScript

Халиманенков Андрей Сергеевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данной статье рассматриваются преимущества, которые несёт под собой использование TypeScript при разработке приложений на JavaScript. Эта надстройка помогает избежать множества ошибок, связанных с типизацией данных.
Ключевые слова: TypeScript, разработка сайтов, JavaScript, HTML, CSS, UI.

Advantages of using TypeScript instead of JavaScript

Khalimanenkov Andrey Sergeevich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article discusses the advantages of using TypeScript for developing JavaScript applications. This language helps to avoid a lot of errors related to data typing.
Keywords: TypeScript, website development, JavaScript, HTML, CSS, UI.

JavaScript является языком с динамической типизацией. Это означает то, что при объявлении переменной в ней может содержаться число, а затем быть присвоена строка, массив, объект или булево значение. Это порождает огромное количество неразберихи и ошибок при разработке больших приложений. Например, в переменную цены или описания товара, вместо числа или строки, может присвоится объект, содержащий эти значения. TypeScript [1] в свою очередь добавляет поддержку типов, а также интерфейсов. Типами могут являться как number, string, object, array и т.д., а также и смешанные типы. Например, в переменную может быть записано либо число, либо строка. Такая переменная примет в себя перечисленные примитивы, но при попытке присвоить такой переменной объект – появится ошибка. TypeScript не является отдельным языком программирования и используется лишь на этапе написания веб-приложений. Итоговый код будет на чистом JavaScript, а весь синтаксис TypeScript будет удалён при сборке проекта.

Ещё одним преимуществом является поддержка «из коробки» в редакторе Visual Studio Code [2]. Типизация кода обрабатывается в реальном времени, и разработчик сразу видит ошибки.

В этой статье будет рассмотрено использование TypeScript.

Цель исследования – разобрать принципы написания TypeScript кода.

Вопрос использования типизации в различных языках программирования, а также разработка веб-приложений на JavaScript волнует некоторых исследователей и специалистов: И. И. Жульковская [3] и др. рассмотрели преимущества и недостатки статической и динамической типизаций в различных языках программирования. А. И. Легалов [4] и др. Предложили способ добавить статическую систему типов в функционально-потоковую модель параллельных вычислений и разработанный на ее основе язык функционально-потокового параллельного программирования. А. А. Воевода и Д. О. Романников [5] в своей работе рассмотрели особенности интерпретируемых языков программирования (ИЯП) на примере языков Ruby и PHP, которые имеют динамическую типизацию. А. М. Заяц и Н. П. Васильев [6] изложили основы работы с объектной моделью документа, положенной в основу динамического формирования и изменения содержимого HTML страниц, с помощью языка программирования JavaScript и библиотеки jQuery. G. Bierman, M. Abadi и M. Torgersen [7] рассмотрели саму сущность TypeScript, причины его появления и выяснили, что система типов не является статически надежной по своему замыслу. A. Feldthaus и A. Møller [8] провели исследование с проверкой типов в существующих библиотеках JavaScript и нашли 142 ошибки в файлах 10 библиотек.

Самая частая ошибка при разработке на JavaScript [9] – это присвоение объекта или undefined вместо поля этого объекта (рис. 1). Это приводит к ошибочному выводу информации. Это может быть цена, название или описание товаров. В таких случаях это является серьёзной ошибкой для бизнеса, из-за которой можно потерять клиентов и выручку. Но JavaScript также используется и при разработке информационных систем, которые отвечают за важные области жизни человека. Например, медицинские порталы для регистратуры в медицинских учреждениях переводят на web, а значит использование JavaScript (далее «JS») неизбежно. Поэтому весь программный код должен быть строго типизирован во избежание серьёзных ошибок.

До появления TS в код вставляли огромное количество проверок на типы.

```
if (typeof someVariable === 'string') {
  //код программы...
}
```

Такие конструкции кратно усложняли процесс разработки, ведь код становился трудночитаемым. Но с появлением TS от этого отказались.

Рисунок 1 – Рендер объекта вместо его поля

Для работы с TypeScript (далее «TS»), его нужно установить глобально на компьютер через менеджер пакетов npm или yarn с помощью команд «npm install -g typescript» и «$ yarn add typescript» соответственно.

Способы добавления типов и интерфейсов

В TS есть интерфейсы, которые работают по аналогии с интерфейсами в других ЯП. Можно чётко задать структуру объекта. Обозначить явно имена ключей, типы значений в ключах объекта, иерархию внутри и т.д.

Они создаются с помощью ключевого слова «interface»:

```
interface IStringObject {
  [key: string]: string
}
```

Идентичного результата можно добиться и с ключевым словом «type»:

```
type IStringObject = {
  [key: string]: string
}
```

Также можно задать тип через конструкцию Record:

```
type IStringObject = Record<string, string>
```

Эти записи идентичны в своей сути. С единственным отличием в том, что при работе с классами в JS следует использовать именно интерфейсы, т.к. классы получают строгую типизацию через расширение от интерфейса. Чётко обозначить типы с помощью «type» можно лишь для обычных переменных, либо ключей внутри интерфейсов.

Типы можно смешивать или объявлять какие значения может содержать каждый тип:

```
type StrOrNum = number | string;
type allOrNone = 'all' | 'none';
```

При попытке присвоить «true» переменной с типом StrOrNum появится ошибка «Type 'boolean' is not assignable to type 'StrOrNum'.», которая говорит о том, что в этом типе не может быть Boolean. Но зато в такую переменную мы можем присвоить строку или число. Схожая ситуация с типом allOrNone. Переменная с этим типом примет только строки ‘all’ или ‘none’ и ничего больше.

Типы переменных

Строка

Если создать переменную со строкой и проверить её тип в чистом JS через команду «typeof», то на выводе получим «string» (рис. 2).

```
> const str = 'a';
  typeof str
<· 'string'
```

Рисунок 2 – встроенная в JS проверка типа строки

В TS тип строки определяется с помощью следующего синтаксиса:

```
let str: string = 'a';
```

При попытке присвоить этой переменной другой тип данных мы получим ошибку (рис. 3), говорящий о том, что тип, который мы хотим присвоить отличается от изначально заданного. Но если мы присвоим переменной str строку ‘b’, то ошибки, ожидаемо, не будет. Такое поведение работает со всеми типами в TS.

```
let str: string
Type 'number' is not assignable to type 'string'. ts(2322)
View Problem    No quick fixes available
str = 2;
```

Рисунок 3 – Ошибка строгой типизации

Число

Если создать переменную с числом и проверить её тип в чистом JS через команду «typeof», то на выводе получим «number» (рис. 4). В JS в этом типе хранятся все числа, такие как float (из других ЯП, например C#), так и int, bigint и т.д.

```
> const num = 2
  typeof num
<· 'number'
```

Рисунок 4 - встроенная в JS проверка типа числа

В TS тип числа определяется с помощью следующего синтаксиса:

```
let num: number = 2;
```

Булев тип

Если создать переменную со булевым значением и проверить её тип в чистом JS через команду «typeof», то на выводе получим «boolean» (рис. 5).

```
> const bool = true;
  typeof bool
<· 'boolean'
```

Рисунок 5 - Встроенная в JS проверка типа булевой переменной

В TS булев тип определяется с помощью следующего синтаксиса:

```
let bool: boolean = true;
```

Объекты

С этим типом больше возможностей в типизации. Для начала посмотрим тип объекта в чистом JS, по аналогии с прошлыми примерами:

Если создать переменную с объектом и проверить её тип в чистом JS через команду «typeof», то на выводе получим «object» (рис. 6).

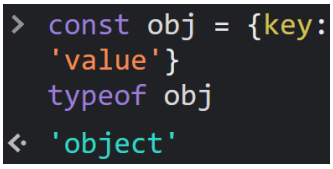

Рисунок 6 - встроенная в JS проверка типа объекта

В TS объект определяется с помощью следующего синтаксиса:

```
let obj: object = {}
```

Но в отличие от предыдущих типов объекты поддерживают больше возможностей в типизации. Пример интерфейса для небольшого списка характеристик автомобиля:

```
interface ICar {
 engine: {
  horsePower: number;
  fuelType: string;
  volume: number;
 };
 transmission: string;
 seatsNumber: number;
}
```

Интерфейсы присваиваются переменным также, как и типы (рис. 7).

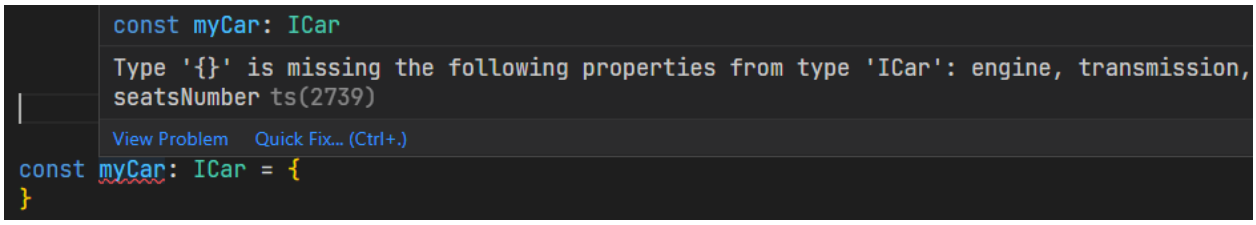

Рисунок 7 – TS уведомляет о том, что в переменной myCar не хватает ключей объекта, объявленных в ICar.

Также, при попытке добавить свойства к такому объекту, появится ошибка (рис. 8)

```
const myCar: ICar = {
  engine: {
    horsePower: 200,
    fuelType: `gas`,
    volume: 1999,
  },
  transmission: `manual`,
  seatsNumber: 2,
}

any
Property 'isElectric' does not exist on type 'ICar'. ts(2339)
View Problem  Quick Fix... (Ctrl+.)
myCar.isElectric = true;
```

Рисунок 8 – Ошибка при добавлении поля, отсутствующего в интерфейсе ICar

Также бывают случаи, когда надо, чтобы в объекте хранились данные только одного типа. Сделать это можно тремя способами:

```
interface IStringObject {
  [key: string]: string
}

type IStringObject = {
  [key: string]: string
}

type IStringObject = Record<string, string>
```

Также у каждого объекта могут быть необязательные поля. Они объявляются следующим образом:

```
interface IObject {
  requiredKey: string;
  notRequiredKey?: string;
}
```

При объявлении объекта с единственным полем «requiredKey» ошибки не будет, т.к. «notRequiredKey» помечен вопросом, а значит является необязательным.

Массивы

Если создать переменную с массивом и проверить её тип в чистом JS через команду «typeof», то на выводе получим «object» (рис. 9).

```
> const arr = [];
  typeof arr
< 'object'
```

Рисунок 9 - встроенная в JS проверка типа массива

И нет, это не ошибка с точки зрения JS. Но ошибка с точки зрения логики разработчика. Поэтому в TS есть тип «Array».

В TS массив определяется следующими вариантами:

```
const stringArr: Array<string> = ['a', 'в'];
const stringArr2: string[] = ['a', 'в'];
const numberString: Array<number> = [1, 2];
const numberString2: number[] = [1, 2];
```

В этом примере только два типа, которыми могут являться элементы массива, но в него можно определить любой существующий тип.

Если же нам нужен привычный для JS массив, то он объявляется следующим образом:

```
const array: any[] = [123, 'awd'];
```

Тип any даёт понять, что в массиве может быть любой тип.

Таким образом в статье был разобран базовый синтаксис TS и примеры его использования. Эта надстройка, можно сказать, является обязательной к изучению любому программисту, использующему JS. TS помогает избежать ошибок, а также подсказывает другим разработчикам как должны выглядеть объекты и какие поля конкретных типов содержать в себе. Этот инструмент легко изучается и дружелюбен к разработчикам, т.к. его ошибки явно дают понять, что не так с текущим кодом и не требует долгого чтения документации или поиска в поисковых системах. Ко всему перечисленному TS даёт специалистам отличный autocomplete, который подсказывает какие поля должны быть в объектах.

Библиографический список

1. TypeScript URL: https://www.typescriptlang.org/ (дата обращения: 01.02.2022).
2. Visual Studio Code URL: https://code.visualstudio.com/ (дата обращения: 01.02.2022).
3. Жульковская И. И., Жульковский О. А., Бильо В. В. Типизация современных языков программирования //Збірник наукових праць Дніпровського державного технічного університету. Технічні науки. 2017. №. 1. С. 143-147.
4. Легалов А. И., Легалов И. А., Матковский И. В. Добавление статической типизации в язык функционально-потокового параллельного программирования //Электронные библиотеки. 2020. Т. 23. №. 4. С. 788-807.
5. Воевода А. А., Романников Д. О. Особенности интерпретируемых языков программирования с динамической типизацией и их влияние на процесс анализа //Сборник научных трудов Новосибирского государственного технического университета. 2014. №. 2. С. 99-106.
6. Заяц А. М., Васильев Н. П. Проектирование и разработка WEB-приложений. Введение в frontend и backend разработку на JavaScript и node. js. 2019.
7. Bierman G., Abadi M., Torgersen M. Understanding typescript //European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 2014. С. 257-281.
8. Feldthaus A., Møller A. Checking correctness of TypeScript interfaces for JavaScript libraries //Proceedings of the 2014 ACM International Conference on

Object Oriented Programming Systems Languages & Applications. 2014. С. 1-16.
9. JavaScript URL: https://ru.wikipedia.org/wiki/JavaScript (дата обращения: 01.02.2022).
10. NPM TypeScript URL: https://npmjs.com/package/typescript (дата обращения: 01.02.2022).
11. Yarn TypeScript URL: https://yarnpkg.com/package/typescript (дата обращения: 01.02.2022).