TypeScript 1

Introduction

TypeScript - what is it

TypeScript - язык программирования, надмножество JavaScript.

Создан компанией Microsoft и выпущен в 2012 году для обеспечения дополнительной статической проверки типов.

TypeScript является обратно совместимым с JavaScript и компилируется в него же. Любой код, корректный для JavaScript, будет корректен и для TypeScript.

TypeScript проверяет типы во время компиляции и выдает ошибку, если переменной когда-либо будет присвоено значение другого типа. При этом ошибка не препятствует выполнению кода.

First steps

Установить TypeScript можно с помощью менеджера пакетов, например, npm:

```
npm install -g tsc
```

Флаг -g означает глобальную установку пакета.

Чтобы скомпилировать ts файл в js используется команда tsc <file>. Пример: файл example.ts

```
function greet(person, date) {
  console.log(`Hello, ${person}! Today is ${date}.`)
}

greet('John')
```

```
tsc example.ts
```

```
Expected 2 arguments, but got 1.
```

После компиляции с ошибкой рядом с example.ts всё же создался файл example.js. TypeScript считает Вас умнее себя, поэтому производит компиляцию даже при наличии ошибок. Изменить данное поведение можно с помощью флага --noEmitOnError. Он дает компилятору знать, что файл с ошибками компилировать НЕ нужно.

Explicit types

Модифицируем предыдущий пример, явно указав типы параметров функции:

```
function greet(person: string, date: Date) {
  console.log(`Hello, ${person}! Today is ${date.toDateString()}.`)
}

greet('John', Date())
```

Мы столкнемся со следующей ошибкой: Argument of type 'string' is not assignable to parameter of type 'Date'. Вызов Date() возвращает строку. Для того, чтобы получить объект Date, следует вызвать new Date().

В большинстве случаев TS в состоянии самостоятельно вывести тип:

```
const msg = 'Hello!'
// const msg: string
```

Downleveling

Понижение уровня кода (downleveling) - процесс преобразования кода с использованием более старой версии языка.

TypeScript по умолчанию преобразует код в ES3.

Скомпилируем написанный ранее код с помощью команды tsc:

```
'use strict'
function greet(person, date) {
  console.log('Hello ' + person + '! Today is ' + date.toDateString() + '.')
}
greet('John', new Date())
```

Шаблонные литералы (или шаблонные строки) были представлены в ES6, поэтому при компиляции она превратилась в объединение строк.

Для изменения спецификации, которой должен соответствовать компилируемый код, используется флаг --target. Например, использование флага --target с аргументом es2015 оставит код из примера неизменным.

Strictness

Строгость проверок, выполняемых TypeScript, возможно настроить. Ее параметры определяются следующими флагами:

1. strict - максимальную строгость.
2. noImplicitAny - когда ts не может самостоятельно вывести тип значения, ему автоматически присваивается any (переменная может обладать любым значением). Использование флага запрещает подобное поведение и приводит к ошибке.
3. strictNullChecks - по умолчанию значения null и undefined могут быть присвоены любому другому типу. Использование флага исключает такую возможность и делает обработку null и undefined более явной.

Config

До текущего момента было рассмотрено несколько флагов, определяющих поведение компилятора. Они представляют лишь малую часть параметров, которыми можно настроить TypeScript проект.

Наличие конфигурационного файла tsconfig.json в каталоге указывает на то, что этот каталог является корнем проекта TypeScript. Этот файл определяет корневые файлы и опции компилятора, необходимые для компиляции проекта.

При вызове tsc без указания файлов, компилятор ищет файл tsconfig.json, начиная с текущего каталога и далее по цепочке родительских каталогов. Указать путь до конфигурационного файла можно с помощью флага --project (-p).

Если при вызове tsc будут указаны файлы, tsconfig.json будет проигнорирован.

Basics

Primitives

В js часто используются 3 следующих примитива: string, number и boolean. В ts для каждого из них существует соответствующий тип:

- string - строковые значения, например, 'Hello World'
- number - числа, например, 42 или 3.14.
- boolean - булевы значения: true и false.

Нельзя путать их с String, Number и Boolean (с большой буквы). Они ссылаются на специальные встроенные типы и редко используются в коде.

Arrays

Для определения массива элементов типа T используется следующий синтаксис: T[].

Также корректным является следующее объявление типа: Array<T>.

Приведенные выше записи являются эквивалентными.

Следующая запись является типом кортежа (tuple): [T]. Он означает буквально массив, состоящий из одного элемента типа T. Аналогично тип [string, number, boolean] является типом массива, состоящего из трех элементов: строки, числа и булева значения.

any

Ts предоставляет специальный тип any. Он служит для отключения проверки типов:

```
let obj: any = { x: 0 }
// Ни одна из строк ниже не приведет к возникновению ошибки на этапе компиляции
obj.foo()
obj()
obj.bar = 100
obj = 'hello'
const n: number = obj
```

Этот тип может быть полезен в случае, когда мы хотим избежать длинного определения типа и уверены в написанном коде. Злоупотреблять им не стоит

Functions

TypeScript также позволяет определять типы для функций.

При определении функции мы можем указать, какие параметры она принимает:

```
function greet(name: string) {
  console.log(`Hello, ${name.toUpperCase()}!`)
}
```

```
greet(42)
// Argument of type 'number' is not
// assignable to parameter of type 'string'.
```

И какое значение возвращает:

```
function getFavouriteNumber(): number {
  return 42
}
```

```
function getFavouriteNumber(): number {
  return '42'
} //Type 'string' is not assignable to type 'number'.
```

В большинстве случает возвращаемый тип выводится автоматически.

Количество аргументов будет проверяться даже без указания типов параметров.

Anonymous functions

TypeScript достаточно умный, чтобы самостоятельно определять типы параметров анонимных функций:

```
const names = ['Alice', 'Bob', 'John']

names.forEach((s) => {
  console.log(s.toUppercase())
})
```

Пусть в вышенаписанном коде и не указано ни одного типа, он является полностью типобезопасным: тип names выводится автоматически - это массив строк (string[]). Далее, на основании значения этой переменной выводится тип и для параметра анонимной функции s - string. Поэтому на s можно вызвать строковый метод toUppercase().

Подобный автовывод типа называется contextual typing.

object

Объектный тип - любое значение со свойствами. Чтобы определить такой тип, достаточно просто перечислить каждое свойство объекта и обозначить его тип. Пример: функция, принимающая объект координат:

```
function printCoords(pt: { x: number; y: number }) {
  console.log(pt.x)
  console.log(pt.y)
}

printCoords({ x: 3, y: 7 })
```

Для разделения свойств можно использовать , или ;.

Тип свойства является опциональным. Свойство без явно определенного типа будет иметь тип any.

Optional properties

Объект может также содержать опциональные свойства. Для обозначения таких свойств используется символ вопроса “?”:

```
function printName(obj: { first: string; last?: string }) {
  // ...
}
printName({ first: 'John' })
printName({ first: 'Jane', last: 'Air' })
```

В JS при обращении к несуществующему свойству возвращается undefined По этой причине, при чтении опционального свойства необходимо выполнять соответствующую проверку или использовать оператор опциональной последовательности (?.):

```
function printName(obj: { first: string; last?: string }) {
  if (obj.last !== undefined) {
    console.log(obj.last.toUpperCase())
  }
  console.log(obj.last?.toUpperCase())
}
```

Unions

Union или объединение - тип, сформированный из 2 и более типов и представляющий значение, которое может иметь один из этих типов.

Типы, входящие в объединение, называются членами (members) объединения.

```
function printId(id: number | string) {
  console.log(`ID: ${id}`)
}

printId(42)
printId('42')
```

TS позволяет делать только такие вещи, которые являются валидными для каждого члена объединения. Для использования конкретного типа из объединения необходимо произвести сужение (narrowing) типа:

```
function printId(id: number | string) {
  console.log(typeof id === 'string' ? id.toUpperCase() : id)
}
```

Type aliases & interfaces

Если тип используется в нескольких местах, его можно переиспользовать. Для этих целей служат синонимы типов (type aliases) и интерфейсы (interfaces):

```ts
type Point = {
  x: number
  y: number
}

function printCoords(pt: Point) {
  console.log(`Значение координаты 'x': ${pt.x}`)
  console.log(`Значение координаты 'y': ${pt.y}`)
}
```

```ts
interface Point {
  x: number
  y: number
}

function printCoords(pt: Point) {
  console.log(`Значение координаты 'x': ${pt.x}`)
  console.log(`Значение координаты 'y': ${pt.y}`)
}
```

TypeScript производит анализ по структуре типов. Даже если типы имеют разные названия, при этом обладая идентичной структурой, для TypeScript они будут равны.

Type vs Interface

Синонимы типов и интерфейсы очень похожи. Почти все возможности interface доступны в type. Ключевым отличием между ними является то, что type не может быть расширен. Чтобы создать тип на основе другого, используются пересечения (intersection):

```ts
interface Animal {
  name: string
}

interface Bear extends Animal {
  honey: boolean
}
```

```ts
type Animal {
  name: string
}

type Bear = Animal & {
  honey: boolean
}
```

Кроме того, при повторном объявлении интерфейса с другими полями произойдет расширение первого:

```ts
interface Window {
  title: string
}

interface Window {
  ts: TypeScriptAPI
}
```

=

```ts
interface Window {
  title: string
  ts: TypeScriptAPI
}
```

```ts
type Window = {
  title: string
}

type Window = {
  ts: TypeScriptAPI
}
```

= Ошибка

Type assertion

TypeScript не всегда может достаточно точно вывести тип. Например, когда мы используем document.getElementById, TS знает лишь то, возвращается HTMLElement, но не знает, какой именно. В таком случае, если мы знаем, что будет возвращен элемент типа, например, HTMLCanvasElement, для большей конкретики можно использовать утверждение типа:

```
const myCanvas = document.getElementById('main_canvas') as HTMLCanvasElement
```

Существует и альтернативный синтаксис:

```
const myCanvas = <HTMLCanvasElement>document.getElementById('main_canvas')
```

TypeScript разрешает утверждения типов, когда считает их корректными. Следующее утверждение приведет к ошибке: `const x = 'hello' as number`

Если же мы уверены, что производим корректное приведение типа, можно использовать двойное утверждение: `const a = expr as any as T`

Literal types

В дополнение к примитивным типам, мы также можем использовать их более узконаправленные версии:

```
type S = 'some string'
type N = 42
type B = true
```

При использовании констант тип будет автоматически сужаться до литерального:

```
const a = 'str'
// typeof a = 'str'
```

Литеральные типы могут быть полезны, например, при создании функции, принимающей значение из ограниченного набора значений:

```
function printText(s: string, alignment: 'left' | 'right' | 'center') {
  // ...
}
printText('Hello World', 'left')
printText("G'day, mate", 'centre')
```

Для преобразования примитивного типа в литеральный используется утверждение типа as const:

```
let s = 'str' as const
```

null & undefined

В JS существует два примитивных значения, сигнализирующих об отсутствии значения: null и undefined. В TypeScript для них существуют соответствующие типы.

Для работы с null и undefined TypeScript предоставляет оператор ненулевого утверждения (non-null assertion operator). Он помечает выражение как ненулевое (исключает из его типа null и undefined):

```
function liveDangerously(x?: number | undefined) {
  // No error!
  console.log(x!.toFixed())
}
```

Control Flow

Control flow analysis

Анализ потока управления - это анализ, который TypeScript выполняет на основе достижимости кода (reachability). Он используется им для сужения типов с учетом защитников типа и присвоений. При анализе переменной поток управления может не раз разделяться, объединяться и ветвиться, поэтому переменная может иметь разные типы в разных участках кода:

```
function example() {
  let x: string | number | boolean
  x = Math.random() < 0.5
  console.log(x)     // boolean
  if (Math.random() < 0.5) {
    x = 'hello'      // string
    console.log(x)
  } else {
    x = 100
    console.log(x)  // number
  }

  return x          // string | number
}
```

typeof operator

Оператор typeof, примененный к выражению, возвращает соответствующее его типу строковое представление. Например, typeof 'hello' вернет 'string'. В TypeScript этот оператор может использоваться для сужения типов:

```
function pow2(x: number | undefined) {
  if (typeof x === 'number') console.log(x * x)
}
```

Также TypeScript учитывает, что выражение typeof null возвращает 'object':

```
function printAll(strs: string[] | null) {
  if (typeof strs === 'object') {
    for (const s of strs) {
      // Error! Object is possibly 'null'.
      console.log(s)
    }
  }
}
```

Truthiness narrowing

В JS мы можем использовать любые выражения в условиях, инструкциях &&, ||, if, приведении к логическому значению с помощью ! и т.д. Эту возможность также можно использовать для сужения типов:

```
function getUserName(user?: User) {
  if (user) {
    return user.name
  }
  return null
}
```

Однако с таким подходом стоит проявлять осторожность, ведь к false приводятся не только null и undefined:

```
function logNumber(num?: number) {
  if (number) {
    console.log(number)
  }
}
```

Данный пример не будет соответствующим образом обрабатывать значение 0.

Также сужать типы возможно проверкой на равенство:

```
function logNumber(num?: number) {
  if (number !== null) {
    console.log(number)
  }
}
```

instanceof

В JS существует оператор instanceof. Он служит для определения факта, является ли сущность экземпляром другой сущности. Этот оператор применяется к значениям, сконструированным с помощью ключевого слова new. В TypeScript этот оператор также может использоваться для сужения типов:

```
function logValue(x: Date | string) {
  if (x instanceof Date) {
    console.log(x.toUTCString())
    // x: Date
  } else {
    console.log(x.toUpperCase())
    // x: string
  }
}
```

Type predicates

Иногда нам необходимо иметь больше контроля над процессом изменения типов. Для таких целей TypeScript предоставляет возможность определять пользовательские защитники типов. Для этого необходимо определить функцию, возвращаемым значением которой является предикат типа:

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

Предикат имеет форму <parameter> is <type>

В данном примере <parameter> - pet, <type> - Fish

Теперь при вызове isFish с необходимой переменной, TypeScript сузит ее тип на основе условия, заданного в теле функции (когда возвращается истина).

TypeScript также исключает тип Fish в ветке else.

```
const pet = getSmallPet()

if (isFish(pet)) {
  pet.swim()
} else {
  pet.fly()
}
```

Discriminated unions

Предположим, у нас стоит задача описать геометрические фигуры - круги и квадраты. Круги работают с радиусом, квадраты - с длиной стороны. Следующий интерфейс может удовлетворить нашим условиям:

```
interface Shape {
  kind: 'circle' | 'square'
  radius?: number
  sideLength?: number
}
```

Однако если мы захотим работать с конкретной фигурой, столкнемся с проблемой: свойства radius и sideLength не являются обязательными. Чтобы решить эту неопределённость, мы можем использовать исключающее объединение. Для этого необходимо разделить Shape на разные интерфейсы и построить на их основе объединение типов:

```
interface Circle {
  kind: 'circle'
  radius: number
}
interface Square {
  kind: 'square'
  sideLength: number
}
type Shape = Circle | Square
```

Теперь свойство kind может использоваться для сужения типа:

```
function getArea(shape: Shape) {
  if (shape.kind === 'circle') {
    return Math.PI * shape.radius! ** 2
  }
}
```

never

В TypeScript существует особый тип never. Он служит для описания состояния, которого не должно существовать, и представляет собой нулевой множество. Этот тип может быть присвоен любому элементу, однако кроме самого never, ему не может быть присвоен никакой другой тип. Это можно использовать для выполнения исчерпывающих проверок. Возьмем Shape из предыдущего примера. Пока объединение состоит из 2 типов, следующий код будет корректен. Однако при добавлении нового типа Triangle произойдет ошибка:

```
function getArea(shape: Shape) {
  switch (shape.kind) {
    case 'circle':
      return Math.PI * shape.radius ** 2
    case 'square':
      return shape.sideLength ** 2
    default:
      const _exhaustiveCheck: never = shape
      return _exhaustiveCheck
  }
}
```

```
type Shape = Circle | Square | Triangle

function getArea(shape: Shape) {
  switch (shape.kind) {
    case 'circle':
      return Math.PI * shape.radius ** 2
    case 'square':
      return shape.sideLength ** 2
    default:
      const _exhaustiveCheck: never = shape
      // Type 'Triangle' is not assignable to type 'never'.
      return _exhaustiveCheck
  }
}
```

unknown

В противовес never в TypeScript существует тип unknown. Этот тип представляет собой любое значение. Он похож на any, но является типобезопасным, поскольку TypeScript не позволяет ничего делать со значениями типа unknown:

```
function f1(a: any) {
  a.b() // OK
}
```

```
function f2(a: unknown) {
  a.b() // Error! Object is of type 'unknown'.
}
```

Этот тип может быть полезен, например, при описании функции парсинга:

```
function safeParse(s: string): unknown {
  return JSON.parse(s)
}

const obj = safeParse(someRandomString)
```

Эта функция принимает на вход строку и возвращает неизвестное значение, которое мы не сможем использовать до выяснения его типа.

Functions

Function type expressions

TypeScript позволяет нам описывать типы функций. Они похожи на стрелочные функции, но вместо возвращаемого значения указывается его тип:

```
function greeter(fn: (a: string) => void) {
  fn('Hello, World')
}
function printToConsole(s: string) {
  console.log(s)
}
greeter(printToConsole)
```

Выражение (a: string) => void описывает функцию с одним параметром типа string, которая ничего не возвращает.

Название параметров при сопоставлении со значением не учитываются, но являются обязательными: тип (string) => void будет описывать функцию с параметром string типа any.

Для типов функций также можно использовать синонимы типов:

```
type GreetFn = (a: string) => void
function greeter(fn: GreetFn) {
  // ...
}
```

Call signatures

В JavaScript функции также могут обладать свойствами. Для описания подобного поведения в TypeScript используются сигнатуры вызовов (call signatures):

```
type DescribableFunction = {
  description: string
  (someArg: number): boolean
}
function doSomething(fn: DescribableFunction) {
  console.log(`${fn.description} returned ${fn(6)}`)
}
```

Такое описание немного отличается по синтаксису от типа-выражения функции: между параметрами и возвращаемым значением место стрелки занимает двоеточие.

Construct signatures

В JavaScript функции могут вызываться с ключевым словом new. TypeScript считает такие функции конструкторами, так как они чаще всего используются для создания объектов. Для определения таких функций существует специальный синтаксис. Он похож на синтаксис сигнатур вызовов, но с ключевым словом new:

```
type SomeConstructor = {
  new (s: string): SomeObject
}
function fn(ctor: SomeConstructor) {
  return new ctor('Hello!')
}
```

Некоторые объекты (например, Date) могут вызываться как с, так и без new. Описать такое поведение тоже возможно:

```
interface CallOrConstruct {
  new (s: string): Date
  (n?: number): number
}
```

Generic functions

Довольно часто в функциях те или иные параметры зависят друг от друга. Пример - функция, возвращающая первый элемент массива:

```
function firstElement(arr: any[]) {
  return arr[0]
}
```

Функция выполняет свою работу, но типом возвращаемого значения является any. Для решения этой проблемы в TypeScript существуют общие типы - дженерики. Общие типы определяются перед параметрами и обрамляются угловыми скобками:

```
function firstElement<Type>(arr: Type[]): Type {
  return arr[0]
}
```

Добавление дженерика Type создало связь между параметром и возвращаемым значением.

Теперь при вызове функции тип возвращаемого значения будет определяться типом параметра:

```
// `s` - `string`
const s = firstElement(['a', 'b', 'c'])
// `n` - `number`
const n = firstElement([1, 2, 3])
```

Constraints

TypeScript позволяет контролировать дженерики с помощью ограничений (constraints). Ограничения описываются с помощью ключевого слова extends:

```
function longest<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a
  } else {
    return b
  }
}

const longerArr = longest([1, 2], [1, 2, 3]) // Ok
const longerStr = longest('alice', 'bob')    // Ok
const longerWat = longest(10, 100)           // Error
```

Введя данное ограничение, мы говорим TS, что функция работает только с типами, имеющими свойство length. Это позволяет использовать свойство в теле функции и приводит к соответствующей ошибке при использовании неверного типа.

Specifying type arguments

TypeScript достаточно умен, чтобы в большинстве случаев делать верные выводы о типах аргументов в вызове дженерика, но так бывает не всегда. Рассмотрим следующую функцию:

```
function combine<Type>(arr1: Type[], arr2: Type[]): Type[] {
  return arr1.concat(arr2)
}
```

Вызов функции с разнотипными массивами приведет к ошибке:

```
const arr = combine([1, 2, 3], ['Hello!'])
// Type 'string' is not assignable to type 'number'.
```

Это происходит потому, что TypeScript определяет Type из первого переданного массива. Решить проблему можно с помощью ручного указания корректного типа:

```
const arr = combine<string | number>([1, 2, 3], ['Hello!'])
```

Optional parameters

Функции в JS могут принимать произвольное количество аргументов. Например, метод toFixed принимает опциональное количество цифр после запятой:

```
function fn(n: number) {
  console.log(n.toFixed())
  console.log(n.toFixed(3))
}
```

Мы можем повторить подобное поведение. Для этого необходимо пометить параметр как опциональный с помощью символа “?”:

```
function f(x?: number) {
  // ...
}
f() // OK
f(10) // OK
```

При определении опциональных параметров к их тип объединяется с undefined. Также допускается явная передача значения undefined. Это будет означать отсутствие аргумента.

Также допускается указание дефолтного значения параметра:

В этом случае объединения типа с undefined не произойдет.

```
function f(x = 10) {
  // ...
}
```

Function overload

Функции могут вызываться с разным количеством аргументов разных типов. Например, функция, принимающая на вход 1 аргумент (timestamp) или 3 аргумента (день, месяц, год) и возвращающая объект Date. Реализовать подобное поведение можно с помощью сигнатур перегрузки (overload signatures). Они указываются перед реализацией функции:

```
function makeDate(timestamp: number): Date
function makeDate(d: number, m: number, y: number): Date
function makeDate(dOrTimestamp: number, m?: number, y?: number): Date {
  if (m !== undefined && y !== undefined) {
    return new Date(y, m, dOrTimestamp)
  } else {
    return new Date(dOrTimestamp)
  }
}
const d1 = makeDate(123456789)
const d2 = makeDate(1, 1, 2022)
```

Сигнатура реализации (implementation signature), не учитывается при вызове и служит только для определения тела.

Сигнатура реализации должна быть совместима с сигнатурами перегрузок.

Function

В TypeScript существует глобальный тип Function. Он описывает такие свойства, как call, bind и apply и другие, характерные для функций в JavaScript, а также имеет специальное свойство, позволяющее вызывать значения этого типа - результатом таких вызовов будет any:

```
function doSomething(f: Function) {
  f(1, 2, 3)
}
```

Подобные вызовы считаются нетипизированными из-за небезопасного возвращаемого типа any. Если необходимо принимать произвольную функцию без ее последующего вызова, предпочтительнее использовать более безопасную альтернативу - () => void.

Rest parameters

В JavaScript существуют функции, принимающие произвольное количество параметров. Их можно определить с помощью синтаксиса остаточных параметров. Они указываются после других параметров и помечаются с помощью оператора “...”:

```
function multiply(n: number, ...m: number[]) {
  return m.map((x) => n * x)
}

const a = multiply(10, 1, 2, 3, 4, 5)   // [10, 20, 30, 40, 50]
```

TypeScript неявно помечает остаточные параметры как any[].

Любая аннотация типа остаточных параметров должна иметь вид Array<Type>, Type[] или являться кортежем.

Rest arguments

В JavaScript также существует синтаксис распространения (spread). Он позволяет передавать произвольное количество элементов массива в качестве параметров. Например, используем метод массива push:

```
const arr1 = [1, 2, 3]
const arr2 = [4, 5, 6]
arr1.push(...arr2)
```

TypeScript не считает массивы иммутабельными, поэтому мы можем столкнуться со следующей проблемой:

```
const args = [8, 5]
const angle = Math.atan2(...args)
// Expected 2 arguments, but got 0 or more.
```

Решением этой проблемы в данной ситуации является использование as const:

```
const args = [8, 5] as const

const angle = Math.atan2(...args)
```