**Frontend questions**

**TypeScript**

https://metanit.com/web/typescript/1.1.php

<https://habr.com/ru/companies/macloud/articles/559976/>

[Medium](https://medium.com/nuances-of-programming/react-typescript-основы-и-лучшие-практики-8d23cb329667)

<https://my-js.org/docs/guide/ts/>

[Questions and answers](https://hashdork.com/ru/популярные-машинописные-вопросы-и-ответы-на-интервью/)

<https://highload.today/otvety-na-samye-populyarnye-voprosy-na-sobesedovanii-po-typescript-svyaz-ts-i-js/>

<https://habr.com/ru/companies/ruvds/articles/419993/>

<https://www.jscamp.app/ru/docs/interview/interview02/>

<https://github.com/FedorovAlexander/typescript-interview-questions-ru>

**Что такое TypeScript?**

TypeScript is a programming language that’s a superset of JavaScript, which means it understands all of JavaScript’s syntax and capabilities, while adding additional features. TypeScript’s primary value add over JavaScript is static typing. This means that type-checking happens at the time of code compilation. Type-checking is a programming language’s way of confirming that all operations have received the correct type of data to perform the operation. For example, a simple addition operation would expect at least two numbers. If one side of the addition operation is not a number, there might be type checking errors or unexpected behavior. Microsoft developed TypeScript and released it to the public in 2012.

TypeScript allows developers to use JavaScript, a language they’re often already familiar with, in a much safer way. Rather than guessing and hoping that code is correct, TypeScript gives immediate in-editor feedback when you’ve made incorrect assumptions about what the code should be doing.

JavaScript is a dynamically typed language, which means that JavaScript code can successfully compile even if there are errors that would prevent the code from running correctly, or at all. The type-checking occurs during runtime when you’re actually executing the code, so the code might seem correct in its file, but then crash or cause odd bugs while it’s running.

In contrast, TypeScript type-checking takes place during compile time, and if there are type errors it will not successfully compile, which prevents bad code from going out to be run.

## **How Does TypeScript Work?**

Browsers and most other platforms like node that run JavaScript do not have native support for TypeScript. TypeScript comes with a compiler called tsc which will take your beautifully crafted TypeScript files, strip all the extras and convert them into JavaScript so that the browser (or whatever platform you’re using) will be able to run your code as if it was JavaScript.

In order to set up tsc, you’ll need to have a tsconfig.json file at the root of your project. It will allow you to specify which version of JavaScript to use when converting your TypeScript files, as well as configure additional TypeScript settings.

## **Why Should I Use TypeScript?**

TypeScript can infer types (even if they’re not explicitly defined by the developer) to power type-checking and autocomplete in a code editor. Autocomplete on its own is very powerful and helps to speed up development and enhance the developer experience. The code editor will be able to easily show what properties exist on an object, what type those properties are, auto complete function and variable names, show syntax highlighting, and much more.

Another way TypeScript makes working in a codebase easier, especially if it’s a large and/or shared codebase, is the way code becomes self documenting. For example, in JavaScript we would have to do a bit of guesswork to determine the type of argument expected in the function below (is birthday a string? A Unix timestamp? number?).

const wishHappyBirthday = (birthday) => {

const today = new Date()

if (today === birthday){

console.log(’Happy birthday!!’)

}

}

Just a note: This function is loosely [pseudocode](https://builtin.com/data-science/pseudocode); in reality, we would do some additional parsing of the date object to extract the month and day.

With TypeScript we can specify the argument type instead. With that simple addition of birthday: Date, we now explicitly know our function is expecting a Date object even if this is the first time we’re seeing this function. Even better, if we accidentally pass in something that is not a Date object, TypeScript will flag the error when we call the function, before we’ve even run our code.

const wishHappyBirthday = (birthday: Date) => {

const today = new Date()

if (today === birthday){

console.log(’Happy birthday!!’)

}

}

## **How to Use TypeScript**

If you’d like to incorporate [TypeScript’s type-checking features into an existing JavaScript project](https://builtin.com/software-engineering-perspectives/create-react-app-typescript), the TypeScript docs [have great suggestions](https://www.typescriptlang.org/) for adding TypeScript features incrementally. The first thing you can do if you’re curious is add the comment // @ts-check to the top of your JavaScript file. This will allow your editor to use TypeScript’s language processing tools to show you any errors right in your code editor’s file. You might think you have perfect code but TypeScript often catches would-be bugs that a code review won’t.

As you move toward total adoption of TypeScript in your project, you’ll find that, while it’s really quite similar to JavaScript, it’s also a bit more constricting (in a good way!). For example, If you instantiate a variable in JavaScript that is a string (let iceCream = “chocolate”), JavaScript will have no problem at all with changing the iceCream variable to a different type (iceCream = true).This can cause unexpected behaviors since it’s essentially impossible to know what to expect when using the iceCream variable.

TypeScript gives us two options for dealing with this problem. The first is to simply let TypeScript infer the type based on the usage. In the let iceCream = “chocolate” example, TypeScript will be able to implicitly infer the iceCream variable should be a string and will not let us later assign a different type to it. The second way is to actually tell TypeScript what type the variable should be (let iceCream: string = “chocolate”). We explicitly assign a type. In both cases, TypeScript will give us an error if we try to reassign this variable to anything but a string.

We can even go so far as to constrict the type further:

let iceCream: “chocolate” | “chocolateChipCookieDough” = “chocolate”

Now we’re telling TypeScript that we will only allow these two specific strings to be assigned to the ice cream variable. TypeScript allows us a lot of customization in the realm of types and this is just a fraction of what you can do.

TypeScript (TS, TScript или «тайпскрипт») — это язык программирования для веб-разработки, основанный на JavaScript. Делает код понятнее и надежнее, добавляет статическую типизацию (переменные привязаны к конкретным типам данных), а также может быть скомпилирован в JavaScript.

[https://highload.today/typescript/#1](https://highload.today/typescript/" \l "1)

Расширенный набор JavaScript, TypeScript компилируется в стандартный JavaScript. С концептуальной точки зрения TypeScript и JavaScript похожи на SASS и CSS.

Другими словами, TypeScript — это JavaScript ES6 с некоторыми дополнительными возможностями. JavaScript — это язык сценариев, более похожий на Python, тогда как TypeScript — это объектно-ориентированный язык со статической типизацией, сравнимый с Java и C#.

Классы и интерфейсы — все это часть объектно-ориентированного дизайна TypeScript, а вывод типов — один из многих инструментов, доступных благодаря его статической типизации.

JavaScript записывается в файл с расширением .js, тогда как TypeScript записывается в файл с расширением .ts.

В отличие от JavaScript, код TypeScript нельзя запустить напрямую в браузере или на любой другой платформе, и браузеры его не понимают. Затем целевая платформа запустит простой JavaScript, созданный из файлов .ts с помощью транспилятора TypeScript.

**Основные компоненты TypeScript?**

The TypeScript language is internally divided into three main layers. Each of these layers is divided into sublayers or components. In the following diagram, we can see the three layers and each of their internal components. These layers are:

1. Language
2. The TypeScript Compiler
3. The TypeScript Language Services



## 1. Language

It features the TypeScript language elements. It comprises elements like syntax, keywords, and type annotations.

## 2. The TypeScript Compiler

The TypeScript compiler (TSC) transform the TypeScript program equivalent to its JavaScript code. It also performs the parsing, and type checking of our TypeScript code to JavaScript code.



Browser doesn't support the execution of TypeScript code directly. So the program written in TypeScript must be re-written in JavaScript equivalent code which supports the execution of code in the browser directly. To perform this, TypeScript comes with TypeScript compiler named "tsc." The current version of TypeScript compiler supports ES6, by default. It compiles the source code in any module like ES6, SystemJS, AMD, etc.

We can install the TypeScript compiler by locally, globally, or both with any **npm** package. Once installation completes, we can compile the TypeScript file by running "tsc" command on the command line.

### Example:

1. $ tsc helloworld.ts   // It compiles the TS file helloworld into the helloworld.js file.

### Compiler Configuration

The TypeScript compiler configuration is given in **tsconfig.json** file and looks like the following:

1. {
2. "compilerOptions": {
3. "declaration": **true**,
4. "emitDecoratorMetadata": **false**,
5. "experimentalDecorators": **false**,
6. "module": "none",
7. "moduleResolution": "node",
8. "noFallthroughCasesInSwitch": **false**,
9. "noImplicitAny": **false**,
10. "noImplicitReturns": **false**,
11. "removeComments": **false**,
12. "sourceMap": **false**,
13. "strictNullChecks": **false**,
14. "target": "es3"
15. },
16. "compileOnSave": **true**
17. }

### Declaration file

When we compile the TypeScript source code, it gives an option to generate a **declaration file** with the extension **.d.ts**. This file works as an interface to the components in the compiled JavaScript. If a file has an extension **.d.ts**, then each root level definition must have the **declare** keyword prefixed to it. It makes clear that there will be no code emitted by TypeScript, which ensures that the declared item will exist at runtime. The declaration file provides IntelliSense for JavaScript libraries like jQuery.

## 3. The TypeScript Language Services

The language service provides information which helps editors and other tools to give better assistance features such as automated refactoring and IntelliSense. It exposes an additional layer around the core-compiler pipeline. It supports some standard typical editor operations like code formatting and outlining, colorization, statement completion, signature help, etc.

<https://www.tutorialspoint.com/typescript/typescript_overview.htm>

**Назовите особенности TypeScript?**

* Статическая типизация — это функция TypeScript, которая помогает проверять типы во время компиляции. Таким образом, даже не выполняя скрипт, вы можете обнаружить ошибки в коде по мере его написания.
* DOM можно изменить с помощью TypeScript для добавления или удаления элементов.
* В любой операционной системе, включая Windows, macOS и Linux, может быть установлен компилятор TypeScript.
* Классы, интерфейсы и модули — вот некоторые из функций, предлагаемых TypeScript. Таким образом, он может создавать объектно-ориентированный код для разработки как на стороне клиента, так и на стороне сервера.
* Большинство функций ECMAScript 2015 (ES 6, 7) уже включены в TypeScript, включая классы, интерфейсы, стрелочные функции и т. д.
* Если вы используете динамическую типизацию JavaScript, TypeScript также поддерживает необязательную статическую типизацию.

## Features of TypeScript

**TypeScript is just JavaScript**. TypeScript starts with JavaScript and ends with JavaScript. Typescript adopts the basic building blocks of your program from JavaScript. Hence, you only need to know JavaScript to use TypeScript. All TypeScript code is converted into its JavaScript equivalent for the purpose of execution.

**TypeScript supports other JS libraries**. Compiled TypeScript can be consumed from any JavaScript code. TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.

**JavaScript is TypeScript**. This means that any valid **.js** file can be renamed to **.ts** and compiled with other TypeScript files.

**TypeScript is portable**. TypeScript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on. Unlike its counterparts, TypeScript doesn’t need a dedicated VM or a specific runtime environment to execute.

### TypeScript and ECMAScript

The ECMAScript specification is a standardized specification of a scripting language. There are six editions of ECMA-262 published. Version 6 of the standard is codenamed "Harmony". TypeScript is aligned with the ECMAScript6 specification.



TypeScript adopts its basic language features from the ECMAScript5 specification, i.e., the official specification for JavaScript. TypeScript language features like Modules and class-based orientation are in line with the EcmaScript 6 specification. Additionally, TypeScript also embraces features like generics and type annotations that aren’t a part of the EcmaScript6 specification.

## Why Use TypeScript?

TypeScript is superior to its other counterparts like CoffeeScript and Dart programming languages in a way that TypeScript is extended JavaScript. In contrast, languages like Dart, CoffeeScript are new languages in themselves and require language-specific execution environment.

The benefits of TypeScript include −

* **Compilation** − JavaScript is an interpreted language. Hence, it needs to be run to test that it is valid. It means you write all the codes just to find no output, in case there is an error. Hence, you have to spend hours trying to find bugs in the code. The TypeScript transpiler provides the error-checking feature. TypeScript will compile the code and generate compilation errors, if it finds some sort of syntax errors. This helps to highlight errors before the script is run.
* **Strong Static Typing** − JavaScript is not strongly typed. TypeScript comes with an optional static typing and type inference system through the TLS (TypeScript Language Service). The type of a variable, declared with no type, may be inferred by the TLS based on its value.
* TypeScript **supports type definitions** for existing JavaScript libraries. TypeScript Definition file (with **.d.ts** extension) provides definition for external JavaScript libraries. Hence, TypeScript code can contain these libraries.
* TypeScript **supports Object Oriented Programming** concepts like classes, interfaces, inheritance, etc.

**Плюсы использования TypeScript?**

* Любой движок JavaScript или браузер может выполнять TypeScript, поскольку он быстрый, простой в освоении и быстро работает.
* Он имеет тот же синтаксис и семантику, что и JavaScript.
* Это облегчает более быстрое написание внешнего кода бэкэнд-разработчиками.
* Он включает в себя возможности ES6 и ES7, которые могут [функция в JavaScript](https://hashdork.com/ru/важные-функции-lodash/) Двигатели ES5, такие как Node.js.
* Существующие библиотеки JavaScript, такие как Jquery, D3.js и т. д., поддерживаются через файл определения с расширением .d.ts.
* Сценарий JavaScript, который уже существует, может вызывать код TypeScript. Он также легко интегрируется с текущими платформами и библиотеками JavaScript.

TypeScript inherits major pros of JavaScript, but also offers additional benefits coming from static typing and other concepts specific to TS. They prove especially useful when it comes to large-sized codebase and distributed teams working on the same project.

### Optional static typing

JavaScript is a dynamically typed language which means that the types are checked, and datatype errors are spotted only at the runtime. Runtime type checking is not, per se, a disadvantage: It offers more flexibility, enabling program components to adapt and change on the fly. But the larger the project and the team, the more undefined variables are added and the more potential mistakes the code amasses.

TypeScript introduces optional strong static typing: Once declared, a variable doesn’t change its type and can take only certain values. The compiler alerts developers to type-related mistakes, so they have no opportunity to hit the production phase. This results in less error-prone code and better performance during execution.

But static typing is not only about catching bugs. It also gives the code more structure, makes it self-documenting and more readable, speeds up debugging and refactoring. All told increases productivity across a large team.

It’s important to note that TS doesn’t force declaring types everywhere. Developers are free to change the level of type strictness in different parts of the project. This approach distinguishes TS from other statically typed languages and allows you to find the right balance between flexibility and correctness.

### Early spotted bugs

Researchers [found](http://earlbarr.com/publications/typestudy.pdf)that TypeScript detects [15 percent](http://ttendency.cs.ucl.ac.uk/) of common bugs at the compile stage. Far from a 100 percent result, this amount is still significant enough to save developers time and let them focus on correcting mistakes in the logic — rather than catching common bugs. Pushing the code through a compiler also decreases the volume of [quality assurance and testing](https://www.altexsoft.com/whitepapers/quality-assurance-quality-control-and-testing-the-basics-of-software-quality-management/) activities.

### Predictability

With TypeScript, everything stays the way it was initially defined. If a variable is declared as a string, it will always be a string and won’t turn into a Boolean. This enhances the likelihood of functions working the way initially intended.

### Readability

Due to adding strict types and other elements that make the code more self-expressive (so-called syntactic sugar), you can see the design intent of developers who originally wrote the code. It’s especially important for distributed teams working on the same project. A code that speaks for itself can offset the lack of direct communication between team members.

### Rich IDE support

Information about types makes editors and Integrated development environments (IDE) much more helpful. They can offer features like code navigation and autocompletion, providing accurate suggestions. You also get feedback while typing: The editor flags errors, including type-related as soon as they occur. All this helps you write maintainable code and results in a significant productivity boost.

With Microsoft Visual Studio as the most popular and natural environment for running TypeScript, it is also supported by many other EDIs including

* [WebStorm](https://www.jetbrains.com/webstorm/), the intelligent JavaScript IDE;
* [Eclipse](https://www.eclipse.org/), an integrated IDE offering a plugin for TS development;
* [Visual Studio Code](https://code.visualstudio.com/), a lightweight cross-platform editor by Microsoft;
* [Atom](https://atom.io/), a cross-platform text editor; and
* [CATS](https://github.com/jbaron/cats), an open source TypeScript development environment.

### Fast refactoring

[Refactoring](https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/) or updating the app without changing its behavior is needed to keep the codebase robust and maintainable. TypeScript makes this important process less painful. With IDEs knowing much about your code, you are equipped with navigation tools like “find all references” or “go to definition.”

Besides, a lot of mistakes are spotted automatically. For example, if you rename a function and later forget to change the name somewhere, TS will alert you to the issue. This simplifies and accelerates refactoring, which is especially beneficial when you deal with large portions of the codebase.

### Power of OOP

TypeScript supports concepts from class-based object-oriented programming (OOP) like classes, interfaces, inheritance, and more. The OOP paradigm facilitates building a well-organized scalable code, and this advantage becomes more obvious as your project grows in size and complexity.

### Cross-platform and cross-browser compatibility

Every device, platform, or browser that runs JavaScript works with TypeScript as well — after the compiler translates it into vanilla JS. Usually, IDEs and editors supporting TypeScript come with a built-in TS compiler ([tsc](http://www.typescriptlang.org/docs/handbook/compiler-options.html" \l "compiler-options)), which can be invoked from the command line. TS allows for converting a part of the codebase or the whole app at once by adding a configuration file called[tsconfig.json](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html) to the proper root directory.

### Huge talent pool

The number of JavaScript developers [hits](https://jaxenter.com/dev-report-javascript-164201.html) 11.4 million — almost twice as much as the number of engineers using Java or Python. Of course, not all of them know TypeScript. But since TS relies on the JS syntax and semantics, the learning curve for JS devs will be smooth. All in all, companies switching to TS will hardly face the talent shortage.

### Support from the tech world

TypeScript is an open-source language with a steadily growing community. The very fact that it is loved by millions of developers and backed by Microsoft conveys confidence that the technology won’t be abandoned, and you always can seek (and get!) assistance from the wide TypeScript community. Many popular software products — such as [Asana](https://asana.com/) and [Slack](https://slack.com/intl/en-ua/?eu_nc=1) — switched to TypeScript to manage and maintain their large codebases.

**Минусы использования TypeScript?**

### **Not true static typing**

Developers mastering C#, C++ or Java often argue that TypeScript is not a true statically typing language. Apart from the fact that this feature is optional for TS, eventually it is transpiled into untyped JavaScript, so there is always the risk of weird type conversions at runtime.

### One more JavaScript to learn

In the previous section we admitted that it would be relatively easy for JS developers to learn TS. However, despite this similarity, they still need to invest time and effort in learning types and some TS-specific constructs. So, if your company plans to move to TypeScript and have JS devs with no previous experience with TS, they won’t hit 100% productivity immediately.

### Bloated code

Enhanced code readability brought by previously mentioned syntactic sugar and type annotations has its downsides. First, you have to write more code and this has the potential of slowing down the development process. Second, additional annotations make TS files larger than those written in plain JS. The difference in the number of code lines depends on numerous factors. The good thing is that all these extra features will disappear after being transpiled. In the end, the browser will execute the plain JavaScript.

### Adding extra step — transpiling

As we wrote before, browsers can’t interpret the TypeScript code, so you need to transpile it to JavaScript before running. However, this process is highly automated and doesn’t require a lot of additional time. In total, the downside of this step is far less significant than its benefits: The compiler spots flaws and flags them before they can cause any damage.

## **It’s getting better all the time**

TypeScript continues to evolve, with its 3.8 version coming soon. At the time of writing, the release candidate was already available, and the production release will come with the same set of upgrades plus some critical bug fixes. So, what major improvements does the fresh version bring?

New syntax for type-only imports and exports. This update is supposed to give developers more control over how the compiler imports and elides (omits) some elements.

Support for ECMAScript private fields. ECMAScript or ES for short is a specification that defines rules and guidelines for scripting languages, and it lies at the core of JS. In 2015, ES introduced the concept of a class to JavaScript — roughly, it’s a template that defines how the function of a certain type (or class) should behave.

By default, ES classes are public, which means they can be accessed and modified outside the class. The implementation of private fields — data slots, available from inside the class only — enables programmers to create a kind of protective cover, secluding internal details and preventing unwelcome interventions. This feature is already supported in Node.js 12, Chrome 74, and Babel.

“Fast and Loose” incremental checking. The new compiling option reveals if the file needs rechecking or rebuilding, based on changes made to it. The update is supposed to reduce build time and thus favor large projects.

Other minor additions and changes will serve the same global idea — to bring productivity, reliability, and predictability to a somewhat chaotic JavaScript world.

<https://blog.logrocket.com/understanding-typescripts-benefits-pitfalls/>

Минусы TypeScript

Невозможно типизировать ошибки, появившиеся в режиме реального времени.

Не все браузеры поддерживают вкладку TypeScript в консоли, что приводит к большим временным затратам.

TypeScript имеет множество нетривиальных классов.

Несмотря на статическую типизацию, можно допустить прохождение любого типа данных, прописав слово any.

<https://blog.skillfactory.ru/glossary/typescript/>

**Типы в TypeScript?**

# <https://metanit.com/web/typescript/2.5.php>

## Типы в TypeScript[​](https://www.jscamp.app/ru/docs/typescript02/" \l "типы-в-typescript)

Все типы в TypeScript являются подтипами главного типа, которым является тип any. Тип any — единственный, который может представлять любое значение JavaScript без всяких ограничений. Все остальные типы накладывают определенные ограничения на свои значения.

### Any[​](https://www.jscamp.app/ru/docs/typescript02/" \l "any)

Тип any используется, когда мы не знаем какой тип должна иметь та или иная переменная. Эта переменная может быть получена из некоего динамического содержимого, как например, если бы использовалась некая сторонняя библиотека. В таких случаях нужно отказаться от проверки типов и позволить таким переменным проходить проверку во время компиляции. Для того, чтобы это сделать, нужно указать переменной тип any с помощью ключевого слова any.

*let* variable: any

[Пример](_blank)

*let* isAny: any

console.log(isAny + ', Тип: ' + *typeof* isAny)

isAny = 'string1'

console.log(isAny + ', Тип: ' + *typeof* isAny)

isAny = 4

console.log(isAny + ', Тип: ' + *typeof* isAny)

isAny = {}

console.log(isAny + ', Тип: ' + *typeof* isAny)

### Boolean[​](https://www.jscamp.app/ru/docs/typescript02/" \l "boolean)

Тип boolean является логическим типом и представлен значениями true и false. Объявляется при помощи ключевого слова boolean.

[Пример](_blank)

*let* isVar1: boolean = false,

isVar2 = true

console.log(isVar1 + ', Тип: ' + *typeof* isVar1)

console.log(isVar2 + ', Тип: ' + *typeof* isVar2)

### Number[​](https://www.jscamp.app/ru/docs/typescript02/" \l "number)

В TypeScript, как и в JavaScript, все производные от number являются 64-битными числами двойной точности с плавающей запятой. Кроме десятичного и шестнадцатеричного формата, поддерживаются бинарный и восьмеричный, введенные в ECMAScript 2015.

[Пример](_blank)

*let* isNumBin: number = 0b00110111, // Binary

isNumOct: number = 0o67, // Octal

isNumDec: number = 55, // Decimal

isNumHex = 0x37 // Hex

console.log(isNumHex + 0xf + ', Тип: ' + *typeof* isNumHex)

### String[​](https://www.jscamp.app/ru/docs/typescript02/" \l "string)

Тип string представляет собой последовательность символов в кодировке Unicode UTF-16. Строки могут быть заключены в одинарные или двойные кавычки, а также в обратные апострофы (инициаторы так называемых шаблонных строк).

[Пример](_blank)

*let* isVar1: string = 'строка',

isVar2 = 'другая строка'

console.log(isVar1 + ', Тип: ' + *typeof* isVar1)

console.log(isVar2 + ', Тип: ' + *typeof* isVar2)

### Symbol[​](https://www.jscamp.app/ru/docs/typescript02/" \l "symbol)

Тип Symbol предоставляет уникальные идентификаторы, которые могут быть использованы как ключи для свойств объекта.

Значения типа symbol создаются с помощью вызова конструктора Symbol.

*let* sym1 = Symbol()

*let* sym2 = Symbol('key') // Необязательный строковый ключ

### Void[​](https://www.jscamp.app/ru/docs/typescript02/" \l "void)

Тип void это нечто противоположное any, то есть отсутствие каких-либо типов. Чаще всего он используется в качестве возвращаемого типа функций, которые не возвращают никакого значения.

[Пример](_blank)

*let* learnTypeScript = (): *void* => {

console.log('Эта функция ничего не возвращает!')

}

console.log(learnTypeScript() + ', Тип: ' + *typeof* learnTypeScript())

### Never[​](https://www.jscamp.app/ru/docs/typescript02/" \l "never)

Тип never представляет тип, значение которого никогда не наступает. Например, never является типом, который возвращает функция, которая всегда бросает исключения или выход из которой никогда не происходит (например, бесконечный цикл). Переменные также могут иметь данный тип, например, для того, чтобы никогда не принимать значение true.

[Пример](_blank)

*let* error = (message: string): never => {

*throw* *new* Error(message)

}

### Null/Undefined[​](https://www.jscamp.app/ru/docs/typescript02/" \l "nullundefined)

В TypeScript оба типа undefined и null фактически имеют имена undefined и null соответственно. Как и void, сами по себе они не особо полезны:

*let* u: *undefined* = *undefined*

*let* n: *null* = *null*

## **Unknown**

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content – e.g. from the user – or we may want to intentionally accept all values in our API. In these cases, we want to provide a type that tells the compiler and future readers that this variable could be anything, so we give it the unknown type.

let notSure: unknown = 4;

notSure = "maybe a string instead";

// OK, definitely a boolean

notSure = false;

[Try](https://www.typescriptlang.org/play/" \l "code/DYUwLgBAdg9mDKBXATiAXBRUDWsDuUEAvBACwDcAULAiiMRAEQC2AhgJ4BG9rEAzmGQBLKAHMIIgSFYATRlUoB6RRADyAaQA0EGSABmIoWBDB2EXpxgxQrKNThJUDPa2B8Q5IA)

If you have a variable with an unknown type, you can narrow it to something more specific by doing typeof checks, comparison checks, or more advanced type guards that will be discussed in a later chapter:

declare const maybe: unknown;

// 'maybe' could be a string, object, boolean, undefined, or other types

const aNumber: number = maybe;

Type 'unknown' is not assignable to type 'number'.Type 'unknown' is not assignable to type 'number'.

if (maybe === true) {

// TypeScript knows that maybe is a boolean now

const aBoolean: boolean = maybe;

// So, it cannot be a string

const aString: string = maybe;

Type 'boolean' is not assignable to type 'string'.Type 'boolean' is not assignable to type 'string'.

}

if (typeof maybe === "string") {

// TypeScript knows that maybe is a string

const aString: string = maybe;

// So, it cannot be a boolean

const aBoolean: boolean = maybe;

Type 'string' is not assignable to type 'boolean'.Type 'string' is not assignable to type 'boolean'.

}

[Try](https://www.typescriptlang.org/play/" \l "code/PTAEAEFMCdoe2gZwFygEwGY1vVnnsAoAE0gGMAbAQ2klDLgDtEAXUAWyoE8AjSVAK6MA1ozgB3RgG5CIUAHJOvSPPpwBFYqD6gqoVtACWjAOYAaUHB4Arciws84cCpCqMLQ0gDNjkYhYRLFgALGFAWLgAHSERCBmY2KgA5AXY+aFRGVPTQAF4Obj4ZQkMvUAAKJR1cmvDoAUgASlAAb0JQUDkAFSjIAGUyI0i2UQlEcOCqNiq6Q3G9R2dXRlAxcXa1BN0AIScXN1RF-ZX8mZkOuT64C0M2MjcxNh09A2MTDfjWXT6WI1NUV6mPIFZQyAC+hBKZXKEWicDKMzytQARICTMjmm0LmAetEBkMRmtxiEpiCdHNdPpfm8PkwvlQfn8TADqUDToVIOdOmArjc7g84E86As9staVsqLslgdtKK3MCzoQwUA)

## Структуры[​](https://www.jscamp.app/ru/docs/typescript02/" \l "структуры)

Обычно в TypeScript не разделяют понятия типа и структуры, потому что структура тоже является типом. Но так как структура – это составной тип данных, который состоит из разных типов, сгруппированных под одним, мы выделили такие типы в отдельную часть.

### Array (массив)[​](https://www.jscamp.app/ru/docs/typescript02/" \l "array-массив)

Тип Array используется для указания элементов массива. Определить массив можно двумя способами:

* []
* Array<elemType>

В первом методе вы указываете тип элементов массива, за которым следует [], который обозначает массив этого типа.

[Пример](_blank)

*let* isArrOfStr: string[] = ['string1', 'string2'],

isArrOfNum: number[] = [1, 2, 3, 4, 5]

console.log(isArrOfNum[4] + ', Тип: ' + *typeof* isArrOfNum)

### Tuple (кортеж)[​](https://www.jscamp.app/ru/docs/typescript02/" \l "tuple-кортеж)

Тип Tuple или кортеж представляет собой массив, каждому из элементов которого можно указать свой тип. Обычно такая структура используется для предоставления набора данных, например, записи в базе данных.

[Пример](_blank)

*let* isTuple: [string, boolean] = ['string1', false]

console.log(isTuple[0] + ', Тип: ' + *typeof* isTuple[0])

console.log(isTuple[1] + ', Тип: ' + *typeof* isTuple[1])

### Enum (перечисление)[​](https://www.jscamp.app/ru/docs/typescript02/" \l "enum-перечисление)

Тип enum используется для объявления перечисления — отдельного типа, который состоит из набора именованных констант, называемого списком перечислителей. Перечисления пришли в TypeScript из C#. Например, для вашего удобства вы можете создать enum дней. По умолчанию первый перечислитель имеет значение 0, и значение каждого последующего перечислителя инкрементируется на единицу.

[Пример](_blank)

*enum* isEnumAnimal {cat, rabbit, horse, dog, hen, cow, sheep}

console.log(isEnumAnimal.rabbit + ', Тип: ' + *typeof* isEnumAnimal.rabbit)

console.log(isEnumAnimal[1] + ', Тип: ' + *typeof* isEnumAnimal[1])

*enum* isEnumWeek {Mon = 34, Tue, Wed, Thu, Fri, Sat, Sun}

console.log(isEnumWeek[34] + ' or ' + isEnumWeek.Tue + ', Тип: ' + *typeof* isEnumWeek)

<https://www.jscamp.app/ru/docs/typescript02/>

<https://www.typescriptlang.org/docs/handbook/basic-types.html>

<https://www.typescriptlang.org/docs/handbook/2/everyday-types.html>

**Что такое декораторы?**

<https://metanit.com/web/typescript/6.1.php>

<https://habr.com/ru/articles/494668/>

Декоратор – приём программирования, который позволяет взять существующую функцию и изменить/расширить её поведение. Декоратор получает функцию и возвращает обёртку, которая делает что-то своё «вокруг» вызова основной функции.

Декоратор — это структурный паттерн, который позволяет добавлять объектам новые поведения на лету, помещая их в объекты-обёртки. Декоратор позволяет оборачивать объекты бесчисленное количество раз благодаря тому, что и обёртки, и реальные оборачиваемые объекты имеют общий интерфейс.

<https://syntaxerror.ru/decorators-in-typescript/>

<https://blog.logrocket.com/practical-guide-typescript-decorators/>

**Поддерживает ли TypeScript перегрузку функций?**

Да

**[Medium](https://medium.com/nuances-of-programming/перегрузка-функций-в-typescript-a2027adadeb1" \l ":~:text=TypeScript спешит на помощь&text=С типизированными методами под рукой,Да!)**

<https://habr.com/ru/companies/otus/articles/688270/>

Перегрузка функций — это возможность определить несколько версий одной функции, каждая из которых принимает свой набор параметров. Разберем на примере:

function concat(a: number, b: number): string;

function concat(a: string, b: string): string;

function concat(a: unknown, b: unknown): string {

if (typeof a === 'number' && typeof b === 'number') {

return `${a.toFixed()}${b.toFixed()}`;

}

return `${a}${b}`;

}

concat('one', 'two'); // onetwo

concat(3, 5.34); // 35

concat(1.33, 10); // 110

Здесь определяется одна функция concat(). У нее две версии, которые выполняют конкатенацию, но делают это по-разному:

* Первая версия — принимает на вход два числа. У чисел сначала отбрасывается дробная часть, потом они конкатенируются
* Вторая версия — принимает на вход две строки. Строки конкатенируются сразу

Реализация поведения для обеих версий делается в третьей функции с тем же именем. При этом описание параметров должно подходить под каждую версию функции. В примере выше типы параметров определены как unknown. Это дает возможность вызывать функцию как со строками, так и с числами.

То, по какой ветке идти, определяется с помощью проверки типов. В примере выше достаточно проверить тип только первого параметра, так как второй в таком случае точно будет строкой. Это обеспечивает система типов и компилятор.

Для перезагрузки не обязательно использовать объявление функций. То же самое можно сделать с помощью стрелочной функции:

const concat: {

(a: number, b: number): string;

(a: string, b: string): string;

} = (a: unknown, b: unknown) => {

if (typeof a === 'number' && typeof b === 'number') {

return `${a.toFixed()}${b.toFixed()}`;

}

return `${a}${b}`;

}

// с использованием алиасов

type Overloaded = {

(a: number, b: number): string;

(a: string, b: string): string;

}

const concat: Overloaded = (a: unknown, b: unknown) {...}

Перегрузка функций не ограничивается двумя версиями. Их может быть сколько угодно. Главное, что в конце всегда описывается функция, которая является общей по параметрам для всех вариантов и внутри которой описывается вся логика для каждого варианта:

function add(a: number, b: number, c: number): number;

function add(a: number, b: number): number;

function add(a: string, b: string): string;

// Сигнатура подходит под все примеры выше

function add(a: unknown, b: unknown, c?: number): unknown {

// тут вся логика

if (c === undefined) {

// ...

}

}

В статических языках перегрузка функций используется достаточно часто, но там она устроена не как в TypeScript. В этих языках создается несколько разных функций, которые с точки зрения программиста имеют одно имя. Поэтому там не нужна общая функция. Логика каждого варианта описывается внутри, что избавляет код от необходимости реализовывать условную логику.

// Пример на Kotlin

fun sayHello (name:String):String {

return "Hi ${name}"

}

fun sayHello (firstName:String, lastName:String):String {

return "Hello ${firstName} ${lastName}"

}

println(sayHello("John", "Doe")) // Hello John Doe

println(sayHello("Elon")) // Hi Elon

Можно задуматься, зачем в TypeScript такая реализация и какие проблемы она решает. Это как и многое другое в TypeScript попытка учесть все возможные варианты написания кода на JavaScript и покрыть их типами для написания типобезопасного кода. В JavaScript нередко создают функции, которые принимают на вход разные типы данных в разных вариациях. Перегрузка функций позволяет описать подобные функции в TypeScript, иначе пришлось бы использовать any и следить за типами самостоятельно.

Технически после транспиляции в JavaScript остается одна функция, как раз та, что содержит тело.

Перегрузка функций в TypeScript — это механизм, который стоит использовать, когда нет другого выбора. В большинстве случаев вместо перегрузки используются объединения или дженерики, о которых мы поговорим позже. Перегрузка нужна, когда между параметрами есть зависимость, например, если оба параметра — строки, либо оба параметра — числа.

<https://www.typescriptlang.org/docs/handbook/2/functions.html>

<https://www.javatpoint.com/typescript-function-overloading>

**Разница между типом (type) и интерфейсом (interface)?**

[https://frontend-stuff.com/blog/types-vs-interfaces/#%D0%BE%D1%82%D0%BB%D0%B8%D1%87%D0%B8%D1%8F](https://frontend-stuff.com/blog/types-vs-interfaces/" \l "отличия)

<https://blog.logrocket.com/types-vs-interfaces-typescript/>

<https://habr.com/ru/sandbox/186102/>

type полезны для объединяемых типов (например, type MyType = TypeA | TypeB ); тогда как интерфейсы лучше использовать для объявления форм, а затем для их реализации или расширения.

Both the methods Type and the Interface are used to describe the structure of the objects in TypeScript. But holds some specific features that will be helpful according to the situation, choosing between them and totally depends on the developer.

Type in TypeScript: The Type System in TypeScript portrays the different data types that are supported by the language. It is divided into three major sections that are Any Type, Built-In Type, and User-Defined Type. The Type System in TypeScript is responsible for checking the data type of any value taken before it can be provided as an input to the program.

Interface in TypeScript: An Interface in TypeScript is a syntactical obligation that all entities must follow. It can only contain the declaration of the members and is responsible for defining the properties, methods, and events. In a way, it is responsible for defining a standard structure that the derived classes will have to follow.

Difference between Type and Interface in TypeScript:

| **Type** | **Interface** |
| --- | --- |
| It is a collection of data types. | It is a form of syntax. |
| Types are more flexible. | Interface is less flexible when compared to typescript types. |
| It uses the “type” keyword for creating new type. | It uses the “interface” keyword for declaring an interface. |
| It supports the creation of a new name for a type. | It provides a way to define the entities. |
| It has less comparatively less capabilities. | It has comparatively more capabilities. |
| It does not support the use of an object. | It supports the use of an object. |
| Multiple merged declarations cannot be used. | Multiple merged declarations can be used. |
| Two types having the same name raise an exception. | Two interfaces having the same name get merged. |
| It does not have implementation purposes. | It has an implementation purpose. |
| It does not support the feature of implementing or extending union types in the class. | It supports the implementation and extend feature in typescript. |
| One can create an intersection type by combining multiple types into a single type. | One cannot create any new intersection interface. |
| It is also used for types such as primitives, unions, and tuples. | It cannot be used with other types of declaration. |

**Что такое JSX в TypeScript? Какие режимы JSX поддерживает TypeScript?**

TypeScript поддерживает транспиляцию JSX и анализ кода. Если вы не знакомы с JSX, вот выдержка с [официального сайта](https://facebook.github.io/jsx/):

JSX - это XML-подобное расширение синтаксиса для ECMAScript без какой-либо определенной семантики. Он НЕ предназначен для имплементации в движках или браузерах. Это НЕ предложение включить JSX в саму спецификацию ECMAScript. Он предназначен для использования различными препроцессорами (транспиляторами) для преобразования этих токенов в стандартный ECMAScript.

Причина использовать JSX состоит в том, чтобы позволить разработчикам писать HTML для пользовательских интерфейсов прямо в JavaScript, чтобы вы могли:

* Проверить типы представления тем же кодом, который проверяет ваш JavaScript.
* Заставить представление знать о контексте, в котором оно будет работать (т.е. укрепить соединение контроллер-представление в традиционном MVC).
* Использовать шаблоны JavaScript для поддержки HTML, например Array.prototype.map, ?:, switch и т.д. Вместо создания новых (и, вероятно, плохо типизированных) альтернатив.

Это снижает вероятность ошибок и повышает удобство поддержки ваших пользовательских интерфейсов. Основным потребителем JSX на данный момент является [ReactJS из Facebook](http://facebook.github.io/react/). Это использование JSX, которое мы обсудим здесь.

# JSX вне React

TypeScript предоставляет вам возможность использовать JSX с проверкой типов в чём-то кроме React. Ниже перечислены параметры настройки, но обратите внимание, что это для продвинутых создателей фреймворков пользовательского интерфейса:

* Вы можете отключить выдачу кода в стиле react, используя опцию "jsx": "preserve". Это означает, что JSX передается как есть, а затем вы можете использовать свой собственный транспилятор для преобразования частей JSX.
* Использование глобального модуля JSX:
  + Вы можете контролировать, какие HTML-теги доступны и как они проверяются на типы, настраивая элементы интерфейса JSX.IntrinsicElements.
  + При использовании компонентов:
    - Вы можете контролировать, какой класс должен быть унаследован компонентами, настроив объявление по умолчанию interface ElementClass extends React.Component<any, any> { }.
    - Вы можете контролировать, какое свойство используется для проверки типов атрибутов (по умолчанию - props), настраивая declare module JSX { interface ElementAttributesProperty { props: {}; } } объявление.

## jsxFactory[¶](https://scriptdev.ru/book/jsx/others/" \l "jsxfactory)

Передача --jsxFactory <имя фабрики JSX> вместе с --jsx react позволяет использовать другую фабрику JSX, отличную от стандартной React.

Новое имя фабрики будет использоваться для вызова функций createElement.

### Пример[¶](https://scriptdev.ru/book/jsx/others/" \l "_1)

import { jsxFactory } from 'jsxFactory';

var div = <div>Hello JSX!</div>;

Скомпилировано с:

tsc --jsx react --reactNamespace jsxFactory --m commonJS

Результат:

'use strict';

var jsxFactory\_1 = require('jsxFactory');

var div = jsxFactory\_1.jsxFactory.createElement(

'div',

null,

'Hello JSX!'

);

## jsx pragma[¶](https://scriptdev.ru/book/jsx/others/" \l "jsx-pragma)

Вы даже можете указать разные jsxFactory для каждого файла, используя jsxPragma, например.

/\*\* @jsx jsxFactory \*/

import { jsxFactory } from 'jsxFactory';

var div = <div>Hello JSX!</div>;

Даже с --jsx react этот файл будет всё равно будет генерировать код учитывая фабрику, указанную в jsx pragma:

'use strict';

var jsxFactory\_1 = require('jsxFactory');

var div = jsxFactory\_1.jsxFactory.createElement(

'div',

null,

'Hello JSX!'

);

JSX - это встраиваемый XML-подобный синтаксис. Он предназначен для преобразования в допустимый JavaScript, хотя семантика этого преобразования зависит от реализации. JSX приобрел популярность с фреймворком React , но с тех пор видел и другие реализации. TypeScript поддерживает встраивание, проверку типов и компиляцию JSX непосредственно в JavaScript.

## Basic usage

Для использования JSX необходимо сделать две вещи.

1. Назовите файлы с расширением .tsx
2. Включите опцию [jsx](https://www.typescriptlang.org/tsconfig" \l "jsx)

TypeScript поставляется с тремя режимами JSX: preserve , react и react-native . Эти режимы влияют только на этап испускания - проверка типов не затрагивается. Режим preserve сохранит JSX как часть вывода для дальнейшего использования на другом этапе преобразования (например, [Babel](https://babeljs.io/) ). Кроме того, файл будет иметь расширение .jsx . В режиме react будет React.createElement , не нужно выполнять преобразование JSX перед использованием, а выходные данные будут иметь расширение файла .js . react-native режим является эквивалентом preserve в том, что он сохраняет весь JSX, но вместо этого вывод будет иметь расширение файла .js .

| Mode | Input | Output | Расширение выходного файла |
| --- | --- | --- | --- |
| preserve | <div /> | <div /> | .jsx |
| react | <div /> | React.createElement("div") | .js |
| react-native | <div /> | <div /> | .js |
| react-jsx | <div /> | \_jsx("div", {}, void 0); | .js |
| react-jsxdev | <div /> | \_jsxDEV("div", {}, void 0, false, {...}, this); | .js |

Вы можете указать этот режим, используя либо [jsx](https://www.typescriptlang.org/tsconfig" \l "jsx)  командной строки jsx, либо соответствующую опцию [jsx](https://www.typescriptlang.org/tsconfig" \l "jsx) в вашем файле tsconfig.json .

<https://www.typescriptlang.org/docs/handbook/jsx.html>  
<https://runebook.dev/ru/docs/typescript/jsx>

TypeScript имеет встроенную поддержку для режимов

preserve , react и react-native.

preserve сохраняет JSX без изменений для использования в последующем преобразовании;

react не выполняет преобразование JSX, а вместо этого генерирует метод react.createElement и выводит его как файл с расширением .js;

react-native сочетает в себе свойства режимов preserve и react: поддерживает весь JSX и выводит его как файл с расширением .js.

**Что такое директивы с тремя наклонными чертами (Triple-Slash Directives), их типы?**

# Triple-Slash Directives

Директивы Triple-slash-это однострочные комментарии,содержащие один XML-тег.Содержимое комментария используется в качестве директив компилятора.

Директивы с тройной косой чертой действительны **только** в верхней части содержащего их файла. Директиве с тройной косой чертой могут предшествовать только однострочные или многострочные комментарии, включая другие директивы с тройной косой чертой. Если они встречаются после оператора или объявления, они рассматриваются как обычные однострочные комментарии и не имеют особого значения.

## /// <reference path="..." />

/// <reference path="..." /> директива является наиболее распространенной из этой группы. Он служит декларацией зависимости между файлами.

Ссылки Triple-slash инструктируют компилятор включать дополнительные файлы в процесс компиляции.

Они также служат методом упорядочивания вывода при использовании [out](https://www.typescriptlang.org/tsconfig" \l "out)  или [outFile](https://www.typescriptlang.org/tsconfig" \l "outFile)  . Файлы отправляются в расположение выходного файла в том же порядке, что и входные, после прохода предварительной обработки.

### Препроцессирование входных файлов

Компилятор выполняет препроцессирование входных файлов для разрешения всех директив ссылки тройной косой черты.В процессе компиляции добавляются дополнительные файлы.

Процесс начинается с набора корневых файлов ; это имена файлов, указанные в командной строке или в списке [files](https://www.typescriptlang.org/tsconfig" \l "files)  в файле tsconfig.json . Эти корневые файлы предварительно обрабатываются в том же порядке, в котором они указаны. Прежде чем файл будет добавлен в список, все ссылки с тройной косой чертой в нем обрабатываются, и их цели включаются. Ссылки с тройной косой чертой разрешаются в первую очередь в глубину в том порядке, в котором они были показаны в файле.

Путь ссылки с тройной косой чертой разрешается относительно содержащего файла,если используется относительный путь.

### Errors

Ошибкой является ссылка на файл,который не существует.Это ошибка,когда в файле есть ссылка на себя с тройной косой чертой.

### Using --noResolve

Если указан флаг компилятора [noResolve](https://www.typescriptlang.org/tsconfig" \l "noResolve)  , ссылки с тройной косой чертой игнорируются; они не приводят ни к добавлению новых файлов, ни к изменению порядка предоставленных файлов.

## /// <reference types="..." />

Подобно директиве /// <reference path="..." /> , которая служит объявлением зависимости , директива /// <reference types="..." /> объявляет зависимость от пакета.

Процесс разрешения этих имен пакетов аналогичен процессу разрешения имен модулей в операторе import . Директивы типов с тройной косой чертой и ссылочными типами проще всего рассматривать как import для пакетов объявлений.

Например, включение /// <reference types="node" /> в файл объявления означает, что этот файл использует имена, объявленные в @types/node/index.d.ts ; и, следовательно, этот пакет необходимо включить в компиляцию вместе с файлом объявления.

Используйте эти директивы только при создании файла d.ts вручную.

Для файлов объявлений, созданных во время компиляции, компилятор автоматически добавит /// <reference types="..." /> для вас; /// <reference types="..." /> в сгенерированный файл декларации добавляется , если и только если полученный файл использует любые объявления из ссылочного пакета.

Для объявить зависимость в @types пакета в .ts файл, используйте [types](https://www.typescriptlang.org/tsconfig" \l "types)  в командной строке или в tsconfig.json вместо этого. См. Дополнительные сведения об [использовании](https://runebook.dev/ru/docs/typescript/tsconfig-json" \l "types-typeroots-and-types)@types , typeRoots и types в файлах tsconfig.json .

## /// <reference lib="..." />

Эта директива позволяет файлу явно включать существующий встроенный файл lib .

На встроенные файлы lib ссылаются так же, как на параметр компилятора [lib](https://www.typescriptlang.org/tsconfig" \l "lib)  в tsconfig.json (например, используйте lib="es2015" , а не lib="lib.es2015.d.ts" и т. Д.).

Для авторов файлов объявлений, которые полагаются на встроенные типы, например, API-интерфейсы DOM или встроенные конструкторы времени выполнения JS, такие как Symbol или Iterable , рекомендуются директивы lib с тройной косой чертой. Раньше в эти файлы .d.ts приходилось добавлять прямые / повторяющиеся объявления таких типов.

Например, добавление /// <reference lib="es2017.string" /> к одному из файлов в компиляции эквивалентно компиляции с --lib es2017.string .

/// <ссылка lib = "es2017.string" />

"foo".padStart(4);

## /// <reference no-default-lib="true"/>

Эта директива отмечает файл как библиотеку по умолчанию . Вы увидите этот комментарий вверху lib.d.ts и его различных вариантов.

Эта директива указывает компилятору не включать библиотеку по умолчанию (например, lib.d.ts ) в компиляцию. Воздействие здесь похоже на передачу [noLib](https://www.typescriptlang.org/tsconfig" \l "noLib)  в командной строке.

Также обратите внимание, что при передаче [skipDefaultLibCheck](https://www.typescriptlang.org/tsconfig" \l "skipDefaultLibCheck)  компилятор пропускает проверку файлов только с /// <reference no-default-lib="true"/> .

## /// <amd-module />

По умолчанию модули AMD генерируются анонимно. Это может привести к проблемам при использовании других инструментов для обработки полученных модулей, таких как сборщики (например, r.js ).

amd-module директива позволяет передавая имя дополнительного модуля для компилятора:

##### amdModule.ts

///<amd-module name="NamedModule"/>

export class C {}

Будет ли в результате присвоения имени NamedModule к модулю как часть вызова AMD define :

##### amdModule.js

define("NamedModule", ["require", "exports"], function (require, exports) {

var C = (function () {

function C() {}

return C;

})();

exports.C = C;

});

## /// <amd-dependency />

**Примечание** : эта директива устарела. Используйте import "moduleName"; заявления вместо этого.

/// <amd-dependency path="x" /> информирует компилятор о зависимости модуля, отличной от TS, которую необходимо внедрить в результирующем вызове require модуля.

amd-dependency директива может также иметь дополнительное name свойства; это позволяет передать необязательное имя для amd-зависимости:

/// <amd-dependency path="legacy/moduleA" name="moduleA"/>

declare var moduleA: MyType;

moduleA.callStuff();

Сгенерированный JS-код:

define(["require", "exports", "legacy/moduleA"], function (

require,

exports,

moduleA

) {

moduleA.callStuff();

});

<https://www.typescriptlang.org/docs/handbook/triple-slash-directives.html>

Директивы с тремя наклонными чертами — это однострочные комментарии, содержащие тег XML для использования в качестве директив компилятора. Каждая директива указывает, что загружать в процесс компиляции. Директивы с тремя наклонными чертами работают только в верхней части своего файла и будут рассматриваться как обычные комментарии в любом другом месте файла.

* /// <reference path=”…” />

  — наиболее распространенная директива, которая определяет зависимость между файлами;

* /// <reference types=”…” />

 — определяет зависимость для пакета;

* /// <reference lib=”…” />

 — позволяет исключить встроенный файл «lib».

**Что такое внешние объявления переменных (ambient declaration) в TypeScript?**

Внешнее объявление переменной (ambient declaration) — это механизм, который позволяет сообщать компилятору TypeScript о том, что некий исходный код существует где-то за пределами текущего файла. Внешние объявления помогают интегрировать в TS-программы сторонние JavaScript-библиотеки.

Эти объявления делают в файле объявления типов с расширением .d.ts. Внешние переменные или модули объявляют так:

declare module Module\_Name {

}

Файлы, в которых находится внешний код, должны быть подключены в TS-файле, использующем их, так:

/// <reference path=" Sample.d.ts"></reference>

[Medium](https://medium.com/@sergey.bakaev/typescript-1-1-внешние-объявления-переменных-ambient-declarations-64177ef2c644)

<https://coderlessons.com/tutorials/veb-razrabotka/izuchite-mashinopis/typescript-ambient>

**Какие элементы ООП поддерживаются в TypeScript?**

TypeScript реализует объектно-ориентированный подход, в нем есть полноценная поддержка классов. Класс представляет шаблон для создания объектов и инкапсулирует функциональность, которую должен иметь объект. Класс определяет состояние и поведение, которыми обладает объект.

Для определения нового класса применяется ключевое слово class. Например, определим класс User:

1

class User {}

После определения класса мы можем создавать его объекты:

let tom: User = new User();

let alice = new User();

Здесь определено два объекта класса User - tom и alice.

Поля класса

Для хранения состояния объекта в классе определяются поля:

class User {

name: string;

age: number;

}

Здесь определены два поля - name и age, которые имеют типы string и number соответственно. Фактически поля представляют переменные уровня класса, только для их объявления не применяются var и let.

По имени объекта мы можем обращаться к этим полям:

class User {

name: string;

age: number;

}

let tom = new User();

tom.name = "Tom";

tom.age = 36;

console.log(`name: ${tom.name} age: ${tom.age}`); // name: Tom age: 36

При определении полей им можно задать некоторые начальные значения:

class User {

name: string = "Tom Smith";

age: number = 18;

}

let user = new User();

console.log(`name: ${user.name} age: ${user.age}`); // name: Tom Smith age: 18

Методы

Классы также могут определять поведение - некоторые действия, которые должны выполнять объекты этого класса. Для этого внутри класса определяются функции, которые называются методами.

class User {

name: string;

age: number;

print(){

console.log(`name: ${this.name} age: ${this.age}`);

}

toString(): string{

return `${this.name}: ${this.age}`;

}

}

Здесь в классе User определены два метода. Метод print() выводит информацию об объекте на консоль, а метод toString() возвращает некоторое представление объекта в виде строки.

В отличие от обычных функций для определения методов не указывается ключевое слово function.

Для обращения внутри методов к полям и другим методам класса применяется ключевое слово this, которое указывает на текущий объект этого класса.

Применение методов:

class User {

name: string;

age: number;

print(){

console.log(`name: ${this.name} age: ${this.age}`);

}

toString(): string{

return `${this.name}: ${this.age}`;

}

}

let tom = new User();

tom.name = "Tom";

tom.age = 36;

tom.print(); // name: Tom age: 36

console.log(tom.toString()); // Tom: 36

Конструкторы

Кроме обычных методов классы имеют специальные функции - конструкторы, которые определяются с помощью ключевого слова constructor. Конструкторы выполняют начальную инициализацию объекта. Например, добавим в класс User конструктор:

class User {

name: string;

age: number;

constructor(userName: string, userAge: number) {

this.name = userName;

this.age = userAge;

}

print(){

console.log(`name: ${this.name} age: ${this.age}`);

}

}

let tom = new User("Tom", 36);

tom.print(); // name: Tom age: 36

Здесь конструктор применимает два параметра и использует их значения для установки значения полей name и age:

constructor(userName: string, userAge: number) {

this.name = userName;

this.age = userAge;

}

Затем при создании объекта в конструктор передается два значения для его параметров:

let tom = new User("Tom", 36);

Поля для чтения

Полям класса в процессе работы программы можно присваивать различные значения, которые соответствуют типу полей. Однако TypeScript также позволяет определять поля для чтения, значения которых нельзя изменить (кроме как в конструкторе). Для определения таких полей применяется ключевое слово readonly:

class User {

readonly name: string = "Default user";

age: number;

constructor(userName: string, userAge: number) {

this.name = userName;

this.age = userAge;

}

print(){

console.log(`name: ${this.name} age: ${this.age}`);

}

}

Значение полей для чтения можно установить либо при объявлении:

readonly name: string = "Default user";

Либо в конструкторе:

constructor(userName: string, userAge: number) {

this.name = userName;

this.age = userAge;

}

В остальных местах программы значение этого поля нельзя изменить. Например, в следующем случае мы получим при компиляции ошибку, потому что пытаемся установить поле для чтения:

class User {

readonly name: string = "Default user";

age: number;

constructor(userName: string, userAge: number) {

this.name = userName;

this.age = userAge;

}

print(){

console.log(`name: ${this.name} age: ${this.age}`);

}

}

let tom = new User("Tom", 36);

tom.name = "Bob"; // ! Ошибка - поле name - только для чтения

tom.print(); // name: Tom age: 36

Еще наследование, абстрактные классы, методы и поля, модификаторы доступа, геттеры и сеттеры, статические поля и методы, интерфейсы, дженерики и тд. [См. https://metanit.com/web/typescript/3.1.php](https://metanit.com/web/typescript/3.1.php)

<https://vc.ru/dev/423888-krupnyy-gayd-po-typescript>

<https://habr.com/ru/companies/nix/articles/301002/>

**Модификаторы доступа в TypeScript?**

Модификаторы доступа позволяют сокрыть состояние объекта от внешнего доступа и управлять доступом к этому состоянию. В TypeScript три модификатора: public, protected и private.

Если к свойствам и функциям классов не применяется модификатор, то такие свойства и функции расцениваются как будто они определены с модификатором public. То есть следующее определение класса:

class Person {

name: string;

year: number;

}

Будет эквивалентно:

class Person {

public name: string;

public year: number;

}

private

Если же к свойствам и методам применяется модификатор private, то к ним нельзя будет обратиться извне при создании объекта данного класса.

Например, создадим класс с приватными свойствами и методами:

class Person {

private \_name: string;

private \_year: number;

constructor(name: string, age: number) {

this.\_name = name;

this.\_year = this.setYear(age);

}

public print(): void {

console.log(`Имя: ${this.\_name} Год рождения: ${this.\_year}`);

}

private setYear(age: number): number {

return new Date().getFullYear() - age;

}

}

let tom = new Person("Tom", 24);

tom.print();

// console.log(tom.\_name); // нельзя обратиться, так как \_name - private

// tom.setYear(45); // нельзя обратиться, так как функция - private

Два свойства \_name и \_year используются с модификатором private, поэтому мы не можем их использовать вне класса, например, console.log(tom.\_name). То же самое относится к функции setYear(). Остальные функции доступны.

protected

Модификатор protected определяет поля и методы, которые из вне класса видны только в классах-наследниках:

class Person {

protected name: string;

private year: number;

constructor(name: string, age: number) {

this.name = name;

this.year = this.setYear(age);

}

protected printPerson(): void {

console.log(`Имя: ${this.name} Год рождения: ${this.year}`);

}

private setYear(age: number): number {

return new Date().getFullYear() - age;

}

}

class Employee extends Person {

protected company: string;

constructor(name: string, age: number, company: string) {

super(name, age);

this.company = company;

}

public printEmployee(): void {

//console.log("Year: " + this.year); // поле year недоступно, так как private

// setYear(25); // метод setYear недоступен, так как private

this.printPerson(); // метод printPerson доступен, так как protected

console.log(`Компания: ${this.company}`);

}

}

let sam = new Employee("Sam", 31, "Microsoft");

sam.printEmployee();

В классе Employee будет доступно поле name, а также метод printPerson(), так как они имеют модификатор protected. А вот приватное свойство year и приватный метод setYear() будут недоступны.

Определение полей через конструктор

Использование модификаторов в параметрах конструктора позволяет сократить написание кода. Например, пусть у нас есть следующий класс:

class Person {

private name: string;

private age: number;

constructor(name: string, age: number) {

this.name = name;

this.age = age;

}

printPerson(): void {

console.log(`Имя: ${this.name} Возраст: ${this.age}`);

}

}

Этот класс будет аналогичен следующему:

class Person {

constructor(private name: string, private age: number) { }

printPerson(): void {

console.log(`Имя: ${this.name} Возраст: ${this.age}`);

}

}

Используя модификаторы в параметрах конструктора, нам больше не надо явно создать свойства для этих параметров. Свойства создаются автоматически, называются они по имени параметров и имеют те же модификаторы, что и параметры.

Подобным образом, если мы хотим сделать свойства публичными, то следует использовать модификатор public:

class Person

constructor(public name: string, public age: number) { }

printPerson(): void {

console.log(`Имя: ${this.name} Возраст: ${this.age}`);

}

}

Если необходимо сделать поле для чтения, то к модификатору доступа добавляется модификатор readonly:

class Person {

constructor(private readonly name: string, private age: number) { }

}

**Разница между внутренним (Internal Module) и внешним модулями (External Module)?**

Модуль разработан с идеей организовать код, написанный на TypeScript. Модули в целом делятся на –

* Внутренние Модули
* Внешние Модули

## **Внутренний модуль**

Внутренние модули пришли в более ранней версии Typescript. Это использовалось для логической группировки классов, интерфейсов, функций в один модуль и может быть экспортировано в другой модуль. Эта логическая группировка называется пространством имен в последней версии TypeScript. Таким образом, внутренние модули устарели, вместо этого мы можем использовать пространство имен. Внутренние модули все еще поддерживаются, но рекомендуется использовать пространство имен поверх внутренних модулей.

### **Синтаксис внутреннего модуля (старый)**

module TutorialPoint {

export function add(x, y) {

console.log(x+y);

}

}

### **Синтаксис пространства имен (новый)**

namespace TutorialPoint {

export function add(x, y) { console.log(x + y);}

}

### **JavaScript сгенерированный в обоих случаях одинаков**

var TutorialPoint;

(function (TutorialPoint) {

function add(x, y) {

console.log(x + y);

}

TutorialPoint.add = add;

})(TutorialPoint || (TutorialPoint = {}));

## **Внешний модуль**

Внешние модули в TypeScript существуют для указания и загрузки зависимостей между несколькими внешними **js-** файлами. Если используется только один файл **js** , то внешние модули не имеют значения. Традиционно управление зависимостями между файлами JavaScript выполнялось с помощью тегов скрипта браузера (<script> </ script>). Но это не расширяемо, поскольку очень линейно при загрузке модулей. Это означает, что вместо загрузки файлов один за другим нет асинхронной опции для загрузки модулей. Когда вы программируете js для сервера, например, NodeJ, у вас даже нет тегов скрипта.

Существует два сценария загрузки зависимых **js-** файлов из одного основного файла JavaScript.

* Сторона клиента – RequireJs
* Сторона сервера – NodeJs

### **Выбор загрузчика модуля**

Для поддержки загрузки внешних файлов JavaScript нам нужен загрузчик модулей. Это будет еще одна библиотека **JS** . Для браузера наиболее распространенной библиотекой является RequieJS. Это реализация спецификации AMD (определение асинхронного модуля). Вместо загрузки файлов один за другим AMD может загружать их все отдельно, даже если они зависят друг от друга.

### **Определение внешнего модуля**

При определении внешнего модуля в TypeScript для CommonJS или AMD каждый файл рассматривается как модуль. Поэтому необязательно использовать внутренний модуль с внешним модулем.

Если вы переносите TypeScript из AMD в модульные системы CommonJs, дополнительная работа не требуется. Единственное, что вам нужно изменить, это просто флаг компилятора. В отличие от JavaScript, при переносе с CommonJ на AMD или наоборот возникают накладные расходы.

Синтаксис для объявления внешнего модуля использует ключевые слова «экспорт» и «импорт».

### **Синтаксис**

//FileName : SomeInterface.ts

export interface SomeInterface {

//code declarations

}

Чтобы использовать объявленный модуль в другом файле, используется ключевое слово импорта, как указано ниже. Имя файла указывается только без расширения.

import someInterfaceRef = require(“./SomeInterface”);

### **пример**

Давайте разберемся в этом на примере.

// IShape.ts

export interface IShape {

draw();

}

// Circle.ts

import shape = require("./IShape");

export class Circle implements shape.IShape {

public draw() {

console.log("Cirlce is drawn (external module)");

}

}

// Triangle.ts

import shape = require("./IShape");

export class Triangle implements shape.IShape {

public draw() {

console.log("Triangle is drawn (external module)");

}

}

// TestShape.ts

import shape = require("./IShape");

import circle = require("./Circle");

import triangle = require("./Triangle");

function drawAllShapes(shapeToDraw: shape.IShape) {

shapeToDraw.draw();

}

drawAllShapes(new circle.Circle());

drawAllShapes(new triangle.Triangle());

Команда для компиляции основного файла TypeScript для систем AMD:

tsc --module amd TestShape.ts

При компиляции он сгенерирует следующий код JavaScript для AMD.

### **Файл: IShape.js**

//Generated by typescript 1.8.10

define(["require", "exports"], function (require, exports) {

});

### **Файл: circle.js**

//Generated by typescript 1.8.10

define(["require", "exports"], function (require, exports) {

var Circle = (function () {

function Circle() {

}

Circle.prototype.draw = function () {

console.log("Cirlce is drawn (external module)");

};

return Circle;

})();

exports.Circle = Circle;

});

### **Файл: Triangle.js**

//Generated by typescript 1.8.10

define(["require", "exports"], function (require, exports) {

var Triangle = (function () {

function Triangle() {

}

Triangle.prototype.draw = function () {

console.log("Triangle is drawn (external module)");

};

return Triangle;

})();

exports.Triangle = Triangle;

});

### **Файл: TestShape.js**

//Generated by typescript 1.8.10

define(["require", "exports", "./Circle", "./Triangle"],

function (require, exports, circle, triangle) {

function drawAllShapes(shapeToDraw) {

shapeToDraw.draw();

}

drawAllShapes(new circle.Circle());

drawAllShapes(new triangle.Triangle());

});

Команда для компиляции основного файла TypeScript для систем **Commonjs** :

tsc --module commonjs TestShape.ts

При компиляции он сгенерирует следующий код JavaScript для **Commonjs** .

<https://habr.com/ru/companies/macloud/articles/563722/>

**Как TypeScript поддерживает необязательные и дефолтные параметры в функции?**

## **Необязательные параметры**

Необязательные параметры могут использоваться, когда аргументы не нужно принудительно передавать для выполнения функции. Параметр можно пометить как необязательный, добавив к его имени вопросительный знак. Необязательный параметр должен быть установлен как последний аргумент в функции. Синтаксис объявления функции с необязательным параметром приведен ниже:

function function\_name (param1[:type], param2[:type], param3[:type])

function disp\_details(id:number,name:string,mail\_id?:string) {

console.log("ID:", id);

console.log("Name",name);

if(mail\_id!=undefined)

console.log("Email Id",mail\_id);

}

disp\_details(123,"John");

disp\_details(111,"mary","mary@xyz.com");

* В приведенном выше примере объявляется параметризованная функция. Здесь третий параметр, т.е. mail\_id, является необязательным параметром.
* Если во время вызова функции необязательному параметру не было передано значение, значение параметра устанавливается в значение undefined.
* Функция печатает значение mail\_id, только если аргументу передано значение.

В приведенном выше примере объявляется параметризованная функция. Здесь третий параметр, т.е. mail\_id, является необязательным параметром.

Если во время вызова функции необязательному параметру не было передано значение, значение параметра устанавливается в значение undefined.

Функция печатает значение mail\_id, только если аргументу передано значение.

При компиляции он сгенерирует следующий код JavaScript –

//Generated by typescript 1.8.10

function disp\_details(id, name, mail\_id) {

console.log("ID:", id);

console.log("Name", name);

if (mail\_id != undefined)

console.log("Email Id", mail\_id);

}

disp\_details(123, "John");

disp\_details(111, "mary", "mary@xyz.com");

Приведенный выше код выдаст следующий вывод:

ID:123

Name John

ID: 111

Name mary

Email Id mary@xyz.com

## **Параметры по умолчанию**

Параметры функции также могут быть назначены значения по умолчанию. Однако таким параметрам также могут быть явно переданы значения.

### **Синтаксис**

function function\_name(param1[:type],param2[:type] = default\_value) {

}

**Примечание** . Параметр не может быть объявлен необязательным и заданным по умолчанию одновременно.

### **Пример: параметры по умолчанию**

function calculate\_discount(price:number,rate:number = 0.50) {

var discount = price \* rate;

console.log("Discount Amount: ",discount);

}

calculate\_discount(1000)

calculate\_discount(1000,0.30)

При компиляции он сгенерирует следующий код JavaScript –

//Generated by typescript 1.8.10

function calculate\_discount(price, rate) {

if (rate === void 0) { rate = 0.50; }

var discount = price \* rate;

console.log("Discount Amount: ", discount);

}

calculate\_discount(1000);

calculate\_discount(1000, 0.30);

Его вывод выглядит следующим образом –

Discount amount : 500

Discount amount : 300

* В этом примере объявляется функция *calc\_discount* . Функция имеет два параметра – цена и курс.
* Значение параметра *rate* по умолчанию установлено на *0.50* .
* Программа вызывает функцию, передавая ей только значение параметра price. Здесь значение *ставки* составляет *0,50* (по умолчанию)
* Вызывается та же функция, но с двумя аргументами. Значение *скорости по* умолчанию перезаписывается и устанавливается равным явно переданному значению.

В этом примере объявляется функция *calc\_discount* . Функция имеет два параметра – цена и курс.

Значение параметра *rate* по умолчанию установлено на *0.50* .

Программа вызывает функцию, передавая ей только значение параметра price. Здесь значение *ставки* составляет *0,50* (по умолчанию)

Вызывается та же функция, но с двумя аргументами. Значение *скорости по* умолчанию перезаписывается и устанавливается равным явно переданному значению.

<https://coderlessons.com/tutorials/veb-razrabotka/izuchite-mashinopis/typescript-funktsii>

**Что такое перечисление (enum)?**

Тип enum или перечисление позволяет определить набор именнованных констант, которые описывают определенные состояния.

Для определения перечисления применяется ключевое слово enum. Например, объявим следующее перечисление:

enum Season { Winter, Spring, Summer, Autumn };

Перечисление называется Season и имеет четыре элемента. Теперь используем перечисление:

enum Season { Winter, Spring, Summer, Autumn };

let current: Season = Season.Summer;

console.log(current); // 2

current = Season.Autumn; // изменение значения

Здесь создается переменная current, которая имеет тип Season. При этом консоль браузера выведет нам число 2 - значение константы Season.Summer.

Числовые перечисления

По умолчанию константы перечисления, как в примере выше, представляют числовые значения. То есть это так называемое числовое перечисление, в котором каждой константе сопоставляется числовое значение.

Так, созданное выше в примере перечисление

enum Season { Winter, Spring, Summer, Autumn };

фактически эквивалентно следующему:

enum Season { Winter=0, Spring=1, Summer=2, Autumn=3 };

Хотя мы можем явным образом переопределить эти значения. Так, мы можем задать значение одной константы, тогда значения следующих констант будет увеличиваться на единицу:

enum Season { Winter=5, Spring, Summer, Autumn }; // 5, 6, 7, 8

Либо можно каждой константе задать свое значение:

enum Season { Winter=4, Spring=8, Summer=16, Autumn=32 }; // 4, 8, 16, 32

Также мы можем получить непосредственно текстовое значение:

enum Season { Winter=0, Spring=1, Summer=2, Autumn=3 };

var current: string = Season[2]; // 2 - числовое значение Summer

console.log(current); // Summer

Строковые перечисления

Кроме числовых перечислений в TypeScript есть строковые перечисления, константы которых принимают строковые значения:

enum Season {

Winter = "Зима",

Spring = "Весна",

Summer = "Лето",

Autumn = "Осень"

};

var current: Season = Season.Summer;

console.log(current); // Лето

Смешанные гетерогенные перечисления

Также можно определять смешанные перечисления, константы которых могут числа и строки.

enum Season {

Winter = 1,

Spring = "Весна",

Summer = 3,

Autumn = "Осень"

};

var current: Season = Season.Summer;

console.log(current); // 3

console.log(Season.Autumn); // Осень

Перечисления в функциях

Перечисление может выступать в качестве параметра функции.

enum DayTime {

Morning,

Evening

};

function welcome(dayTime: DayTime){

if(dayTime === DayTime.Morning){

console.log("Доброе утро");

}

else{

console.log("Добрый вечер");

}

}

let current: DayTime = DayTime.Morning;

welcome(current); // Доброе утро

welcome(DayTime.Evening); // Добрый вечер

Каждая константа перечисления описывает некоторое состояние. И функция welcome() в виде параметра dayTime принимает это состояние и в зависимости от полученного значения выводит на консоль определенное значение.

Однако стоит отметить, что поскольку здесь перечисление DayTime представляет числовое перечисление, то в реальности в функцию welcome() мы можем передать числовые значения:

welcome(1); // Добрый вечер

Либо даже определить параметр функции как числовой и передавать константы числового перечисления:

enum DayTime {

Morning,

Evening

};

function welcome(dayTime: number){

if(dayTime === DayTime.Morning){

console.log("Доброе утро");

}

else{

console.log("Добрый вечер");

}

}

let current: DayTime = DayTime.Morning;

welcome(current); // Доброе утро

welcome(DayTime.Evening); // Добрый вечер

Пример параметра-строкового перечисления:

enum DayTimeMessage {

Morning = "Доброе утро",

Evening = "Добрый вечер"

};

function welcome(message: DayTimeMessage){

console.log(message);

}

let mes: DayTimeMessage = DayTimeMessage.Morning;

welcome(mes); // Доброе утро

welcome(DayTimeMessage.Evening); // Добрый вечер

При использовании строковых перечислений в отличие от числовых мы не можем передать переметру произвольную строку:

welcome("Привет, ты спишь?")

В этом случе компилятор выкатит нам ошибку при компиляции.

В то же время если параметр представляет тип string, то такому параметру можно передавать как строки, так и константы строкового перечисления:

enum DayTimeMessage {

Morning = "Доброе утро",

Evening = "Добрый вечер"

};

function welcome(message: string){

console.log(message);

}

let mes: DayTimeMessage = DayTimeMessage.Morning;

welcome(mes); // Доброе утро

welcome(DayTimeMessage.Evening); // Добрый вечер

<https://code-basics.com/ru/languages/typescript/lessons/enums>

**Для чего в TypeScript используется NoImplicitAny?**

### **Строгость[​](https://my-js.org/docs/guide/ts/" \l "строгость)**

Строгость проверок, выполняемых TS, определяется несколькими флагами. Флаг --strict или настройка "strict": true в [tsconfig.json](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html) включает максимальную строгость. Двумя другими главными настройками, определяющими строгость проверок, являются noImplicitAny и strictNullChecks.

* noImplicitAny - когда TS не может сделать точный вывод о типе значения, он присваивает такому значению наиболее мягкий тип any. Данный тип означает, что значением переменной может быть что угодно. Однако, использование данного типа противоречит цели использования TS. Использование флага noImplicitAny или соответствующей настройки приводит к тому, что при обнаружении переменной с неявным типом any выбрасывается исключение
* strictNullChecks - по умолчанию значения null и undefined могут присваиваться любым другим типам. Это может облегчить написание кода в некоторых ситуациях, но также часто приводит к багам, если мы забыли их правильно обработать. Флаг strictNullChecks или соответствующая настройка делает обработку null и undefined более явной и избавляет нас от необходимости беспокоиться о том, что мы забыли их обработать

**Разница между типами “Объединение” (|) и “Пересечение” (&)?**

Чем в TypeScript можно заменить наследование? Как указать сразу диапазон типов? Ответы на эти вопросы вы сможете получить, прочитав до конца данную главу.

## Тип Объединение[¶](https://scriptdev.ru/guide/016/" \l "_1)

Объединение (Union) - это мощный механизм, позволяющий создавать из множества существующих типов логическое условие, по которому данные могут принадлежать только к одному из указанных типов. Объединение указывается с помощью оператора прямой черты |, по обе стороны которой располагаются типы данных.

let v1: T1 | T2 | T3;

Переменной, которой был указан тип объединения A или B или C, может быть присвоено значение, принадлежащее к одному из трех типов.

class A {

a: number;

}

class B {

b: string;

}

class C {

c: boolean;

}

// значение может принадлежать только одному типу (A или B или C)

let identifier: A | B | C;

Поскольку значение может принадлежать ко множеству порой несовместимых типов, компилятор, без вмешательства разработчика, то есть без конкретизации типа, определяет принадлежность значения к типу, который наделен общими для всех типов признаками.

class Bird {

fly(): void {}

toString(): string {

return 'bird';

}

}

class Fish {

swim(): void {}

toString(): string {

return 'fish';

}

}

class Insect {

crawl(): void {}

toString(): string {

return 'insect';

}

}

function move(animal: Bird | Fish | Insect): void {

animal.fly(); // Error

animal.swim(); // Error

animal.crawl(); // Error

animal.toString(); // ок, задекларировано во всех типах

}

## Тип Пересечение (Intersection Type)[¶](https://scriptdev.ru/guide/016/" \l "intersection-type)

Пересечение (Intersection) — ещё один мощный механизм TypeScript, который позволяет рассматривать множество типов данных как единое целое. Пересечение указывается с помощью оператора амперсанда &, по обе стороны которого указываются типы данных.

let v1: T1 & T2 & T3;

Переменной, которой был указан тип пересечение A и B и С, должно быть присвоено значение, принадлежащее к типам A и B и C одновременно. Другими словами, значение должно обладать всеми обязательными признаками каждого типа, определяющего пересечение.

class A {

a: number;

}

class B {

b: string;

}

class C {

c: boolean;

}

let name: A & B & C; // значение должно принадлежать ко всем типам одновременно

## **Объединения (unions)**

Обратите внимание: в литературе, посвященной TS, union, обычно, переводится как объединение, но фактически речь идет об альтернативных типах, объединенных в один тип.

### Определение объединения

Объединение — это тип, сформированный из 2 и более типов, представляющий значение, которое может иметь один из этих типов. Типы, входящие в объединение, называются членами (members) объединения.

Реализуем функцию, которая может оперировать строками или числами:

function printId(id: number | string) {

console.log(`Ваш ID: ${id}`)

}

// OK

printId(101)

// OK

printId('202')

// Ошибка

printId({ myID: 22342 })

// Argument of type '{ myID: number }' is not assignable to parameter of type 'string | number'. Type '{ myID: number }' is not assignable to type 'number'. Аргумент типа '{ myID: number }' не может быть присвоен параметру типа 'string | number'. Тип '{ myID: number }' не может быть присвоен типу 'number'

### Работа с объединениями

В случае с объединениями, TS позволяет делать только такие вещи, которые являются валидными для каждого члена объединения. Например, если у нас имеется объединение string | number, мы не сможем использовать методы, которые доступны только для string:

function printId(id: number | string) {

console.log(id.toUpperCase())

// Property 'toUpperCase' does not exist on type 'string | number'. Property 'toUpperCase' does not exist on type 'number'.

}

Решение данной проблемы заключается в сужении (narrowing) объединения. Например, TS знает, что только для string оператор typeof возвращает 'string':

function printId(id: number | string) {

if (typeof id === 'string') {

// В этой ветке `id` имеет тип 'string'

console.log(id.toUpperCase())

} else {

// А здесь `id` имеет тип 'number'

console.log(id)

}

}

Другой способ заключается в использовании функции, такой как Array.isArray:

function welcomePeople(x: string[] | string) {

if (Array.isArray(x)) {

// Здесь `x` - это 'string[]'

console.log('Привет, ' + x.join(' и '))

} else {

// Здесь `x` - 'string'

console.log('Добро пожаловать, одинокий странник ' + x)

}

}

В некоторых случаях все члены объединения будут иметь общие методы. Например, и массивы, и строки имеют метод slice. Если каждый член объединения имеет общее свойство, необходимость в сужении отсутствует:

function getFirstThree(x: number[] | string ) {

return x.slice(0, 3)

}

<https://www.serverlessguru.com/blog/demystifying-typescript-unions-and-intersection>

**Что такое общие типы (generic) в TypeScript?**

<https://www.w3schools.com/typescript/typescript_basic_generics.php>

<https://habr.com/ru/companies/tinkoff/articles/588655/>

Дженерики (generic) помогают писать универсальный, переиспользуемый код, а также в некоторых случаях позволяют отказаться от any. Главная задача дженериков — помочь разработчику писать код, который одинаково будет работать со значениями разных типов.

Посмотрим на примере из реального мира.

Представьте завод по изготовлению автомобилей. Старый завод, который проектировался для сборки автомобиля определённой модели. На нём могут собирать только такую модель автомобиля, а если потребуется выпустить машину с немного другим кузовом, то придётся строить новый завод. Это неоптимальное решение. Если разные машины собираются одинаково то лучше научиться собирать разные машины на одном заводе.

Кажется очевидным, но реальность такова:

❌ Строить новый завод для каждой модели машины

✅ Собирать разные машины на одном заводе

Суть дженериков

С дженериками тоже примерно так. Если мы напишем функцию и жёстко зададим тип, то она сможет работать только со значениями этого типа. Значения других типов передать не получится. Есть два способа это поправить.

Первый способ. Написать несколько одинаковых функций, которые работают с разными типами. Например, такая функция проверяет, есть ли в массиве конкретный элемент.

function includeStr(array: string[], query:string): boolean {

// на входе массив и строка для поиска

for (const value of array) {

// перебираем массив

if (value === query) {

// если в массиве есть элемент — возвращаем true

return true;

}

}

// если ничего не нашлось, возвращаем false

return false;

}

Функция будет отлично работать на массивах из строк. Но для поиска в массиве из чисел придётся дублировать функцию, менять типы, но сам код функции останется неизменным. Например:

function includeNumber(array: number[], query:number): boolean {

// всё то же самое, только на входе числа

for (const value of array) {

if (value === query) {

return true;

}

}

return false;

}

Вот тут на помощь и приходят дженерики. Они помогают написать код, который одинаково работает с данными разных типов.

❌ Пишем много функций для разных типов

✅ Объявляем в функции параметр типа, а потом передаём через него нужный тип

Вместо конкретного типа, мы как будто объявляем «переменную», а затем передаём в неё нужный тип. Таким образом, получается код, который может работать с разными типами:

function include<T>(array: T[], query:T): boolean {

for (const value of array) {

if (value === query) {

return true;

}

}

return false;

}

Код функции не поменялся, но теперь мы не указываем конкретный тип. Мы заводим переменную T и говорим, что тип параметра array — это тип, который будет передан в переменную T. А тип параметра query — это тип, который будет передан через переменную T.

Когда мы захотим воспользоваться этой функцией, то помимо данных для параметров array и query мы ещё должны передать информацию о типах (для переменной T). В первом примере мы передаём тип string, а во втором — number.

// передаём string в качестве типа

include<string>(['igor', 'sasha', 'ira'], 'ira'); // true

// передаём number в качестве типа

include<number>([1, 3, 5], 7); // false

Получается, что с помощью дженериков мы смогли написать код, который работает с разными типами значений. То есть, если коротко.

👉 Дженерики — переменные, через которые мы можем передавать тип.

**Какие области видимости доступны в TypeScript?**

Глобальная: определяется вне любого класса и может использоваться в любом месте программы.  
Область видимости функции / класса: переменные, определенные в функции или классе, могут использоваться в любом месте в пределах этой области.  
Локальная область действия / блок кода: переменные, определенные в локальной области видимости, могут использоваться в любом месте этого блока.

[Еще ответ](https://urvanov.ru/2017/06/03/typescript-объявление-переменных/)

**Что такое .map файл, как и зачем его использовать?**

Это файлы карты, которые позволяют инструментам сопоставлять выпущенный код JavaScript и исходные файлы TypeScript, которые его создали.

[https://futurei.ru/instructions/kak-ispolzovat-ishodnye-karty-javascript-fayly-map/](https://www.w3schools.com/typescript/typescript_basic_generics.php)

**Можно ли использовать TypeScript в серверной разработке?**

Да

Фактически, после компиляции программу на TypeScript можно выполнять в любом современном браузере или использовать совместно с серверной платформой Node.

<https://habr.com/ru/articles/328466/>

**Для чего в TypeScript используют ключевое слово declare?**

Ключевое слово declare используется в TypeScript для объявления переменных, источником которых может служить некий файл, не являющийся TypeScript-файлом.

**Разница между типами void, never и unknown?**

## Void отсутствие конкретного типа[¶](https://scriptdev.ru/guide/014/" \l "void)

Тип данных Void можно назвать полной противоположностью типа any, так как этот тип означает отсутствие конкретного типа. Основное предназначение типа Void — явно указывать на то, что у функции или метода отсутствует возвращаемое значение.

Тип данных Void указывается с помощью ключевого слова void (не путать с одноимённым оператором из JavaScript) и, в отличие от таких типов, как null и undefined, не имеет никаких значений.

Тип void является подтипом any и супертипом для null и undefined.

function action(): void {}

class TypeSystem {

static any: any = action(); // Ok

static number: number = action(); // Error

static string: string = action(); // Error

static boolean: boolean = action(); // Error

static null: null = action(); // Error

static undefined: undefined = action(); // Error

static void: void = action(); // Ok

}

TypeSystem.void = TypeSystem.any; // Ok

TypeSystem.void = TypeSystem.number; // Error

TypeSystem.void = TypeSystem.string; // Error

TypeSystem.void = TypeSystem.boolean; // Error

TypeSystem.void = TypeSystem.null; // Ok

TypeSystem.void = TypeSystem.undefined; // Ok

TypeSystem.void = TypeSystem.void; // Ok

Однако с активным флагом --strictNullChecks, тип данных void совместим лишь с any и undefined.

function action(): void {}

class TypeSystem {

static any: any = action(); // Ok

static number: number = action(); // Error

static string: string = action(); // Error

static boolean: boolean = action(); // Error

static null: null = action(); // Error

static undefined: undefined = action(); // Error

static void: void = action(); // Ok

}

TypeSystem.void = TypeSystem.any; // Ok

TypeSystem.void = TypeSystem.number; // Error

TypeSystem.void = TypeSystem.string; // Error

TypeSystem.void = TypeSystem.boolean; // Error

TypeSystem.void = TypeSystem.null; // Error

TypeSystem.void = TypeSystem.undefined; // Ok

TypeSystem.void = TypeSystem.void; // Ok

Кому-то может показаться, что примеры чересчур излишни, или что примеры, в которых результат вызова функции, не имеющей возвращаемого значения, присваивается полям с различными типами, не имеет никакого отношения к реальности. Да, это так. Но целью данных примеров является научить думать как компилятор TypeScript.

Когда функции в качестве возвращаемого типа указан тип void, может показаться, что возвращая различные значения с помощью оператора return, компилятор выбрасывает ошибки из-за понимания, что функция помечена как ничего не возвращающая. Но это не так. Ошибка возникает по причине несовместимости типов.

function a(): void {

let result: number = 5;

return result; // Error

}

function b(): void {

let result: string = '5';

return result; // Error

}

function c(): void {

let result: any = 5;

return result; // Ok

}

Нельзя не упомянуть, что для функций и методов, которые ничего не возвращают и у которых отсутствует аннотация типа возвращаемого значения, вывод типов определяет принадлежность к типу void.

function action() {

// function action(): void

}

В отличие от null и undefined, тип void не имеет ни одного значения, которое могло бы явно продемонстрировать присвоение. Однако компилятор понимает, что имеет дело с типом void при вызове функции или метода, которые не возвращают значение. Это становится ещё нагляднее, когда вывод типов устанавливает тип, полученный при вызове функции или метода, которые ничего не возвращают.

function action(): void {}

let identifier = action(); // identifier: void

Тип void является уникальным для TypeScript. В JavaScript подобного типа не существует.

## Never примитивный тип[¶](https://scriptdev.ru/guide/014/" \l "never)

Примитивный типа данных Never служит для указания того, что какие-либо операции никогда не будут выполнены.

Never обозначается ключевым словом never и так же, как и void, не имеет явных значений.

Тип данных never является подтипом всех типов, что делает его совместимым со всеми остальными типами.

function action(): never {

throw new Error();

}

class TypeSystem {

static any: any = action(); // Ok

static number: number = action(); // Ok

static string: string = action(); // Ok

static boolean: boolean = action(); // Ok

static null: null = action(); // Ok

static undefined: undefined = action(); // Ok

static void: void = action(); // Ok

static never: never = action(); // Ok

}

TypeSystem.never = TypeSystem.any; // Error

TypeSystem.never = TypeSystem.number; // Error

TypeSystem.never = TypeSystem.string; // Error

TypeSystem.never = TypeSystem.boolean; // Error

TypeSystem.never = TypeSystem.null; // Error

TypeSystem.never = TypeSystem.undefined; // Error

TypeSystem.never = TypeSystem.void; // Error

TypeSystem.never = TypeSystem.never; // Ok

Так как типу never нельзя присвоить значение, отличное от самого типа never, единственным местом, в котором его может использовать разработчик, является аннотация возвращаемого из функции или метода значения, с одной оговоркой. Тип never можно указать только той функции, из которой программа действительно никогда не сможет выйти.

Такой сценарий может выражаться в виде функции, вызов которой приведет к однозначному исключению или тело функции будет включать бесконечный цикл.

function error(message: string): never {

throw new Error(message);

}

function loop(): never {

while (true) {}

}

Вывод типов определит принадлежность возвращаемого функцией значения к типу never, только если он указан в аннотации возвращаемого типа явно.

function error(message: string): never {

throw new Error(message);

}

function action() {

// function action(): never

return error('All very, very bad.');

}

let identifier = error(); // let identifier: never

let identifier = action(); // let identifier: never

Стоит заметить, что без явного указания типа never вывод типов определит принадлежность возвращаемого значения к типу void.

function error(message: string) {

// function error(): void

throw new Error(message);

}

function loop() {

// function loop(): void

while (true) {}

}

Тип never является уникальным для TypeScript. В JavaScript подобного типа не существует.

## Unknown[¶](https://scriptdev.ru/guide/014/" \l "unknown)

Тип Unknown является типобезопасным аналогом типа any и представлен в виде литерала unknown. Все типы совместимы с типом unknown, в то время как сам тип unknown совместим только с самим собой и типом any.

class TypeSystem {

static unknown: unknown;

static any: any = TypeSystem.unknown; // Ok

static number: number = TypeSystem.unknown; // Error

static string: string = TypeSystem.unknown; // Error

static boolean: boolean = TypeSystem.unknown; // Error

static null: null = TypeSystem.unknown; // Error

static undefined: undefined = TypeSystem.unknown; // Error

static void: void = TypeSystem.unknown; // Error

static never: never = TypeSystem.unknown; // Error

}

TypeSystem.unknown = TypeSystem.any; // Ok

TypeSystem.unknown = TypeSystem.number; // Ok

TypeSystem.unknown = TypeSystem.string; // Ok

TypeSystem.unknown = TypeSystem.boolean; // Ok

TypeSystem.unknown = TypeSystem.null; // Ok

TypeSystem.unknown = TypeSystem.undefined; // Ok

TypeSystem.unknown = TypeSystem.void; // Ok

TypeSystem.unknown = TypeSystem.unknown; // Ok

Кроме того, над типом unknown запрещено выполнение каких-либо операций.

let v0: any;

v0.a = 5; // Ok

v0.a = ''; // Ok

v0(); // Ok

let v1: unknown = v0; // Ok

v1.a = 5; // Error

v1.a = ''; // Error

v1(); // Error

Если тип unknown составляет тип пересечение (intersection), то он будет перекрыт всеми типами.

type T0 = any & unknown; // type T0 = any

type T1 = number & unknown; // type T1 = number

type T2 = string & unknown; // type T2 = string

type T3 = boolean & unknown; // type T3 = boolean

type T4 = null & unknown; // type T4 = null

type T5 = undefined & unknown; // type T5 = undefined

type T6 = void & unknown; // type T6 = void

type T7 = never & unknown; // type T7 = never

type T8<T> = T & unknown; // type T8 = T

type T9 = unknown & unknown; // type T9 = unknown

Если тип unknown составляет тип объединение (union), то он перекроет все типы, за исключением типа any.

type T0 = any | unknown; // type T0 = any

type T1 = number | unknown; // type T1 = unknown

type T2 = string | unknown; // type T2 = unknown

type T3 = boolean | unknown; // type T3 = unknown

type T4 = null | unknown; // type T4 = unknown

type T5 = undefined | unknown; // type T5 = unknown

type T6 = void | unknown; // type T6 = unknown

type T7 = never | unknown; // type T7 = unknown

type T8<T> = T | unknown; // type T8 = unknown

type T9 = unknown | unknown; // type T9 = unknown

Помимо этого, запрос ключей (keyof) для типа unknown возвращает тип never.

type T0 = keyof number; // type T0 = "toString" | "toFixed" | "toExponential" | "toPrecision" | "valueOf" | "toLocaleString"

type T1 = keyof any; // type T1 = string | number | symbol

type T2 = keyof unknown; // type T2 = never

Тип unknown позволяется использовать только в операциях равенства ===, ==, !== и != и в операциях с логическими операторами &&, || и !.

let v0: unknown = 5;

let v1 = 5 === v0; // Ok

let v2 = 5 !== v0; // Ok

let v3 = 5 > v0; // Error

let v4 = 5 < v0; // Error

let v5 = 5 >= v0; // Error

let v6 = 5 <= v0; // Error

let v7 = 5 - v0; // Error

let v8 = 5 \* v0; // Error

let v9 = 5 / v0; // Error

let v10 = ++v0; // Error

let v11 = --v0; // Error

let v12 = v0++; // Error

let v13 = v0--; // Error

let v14 = 5 && v0; // Ok, let v14: unknown

let v15 = 5 || v0; // Ok, let v15: number

let v16 = v0 || 5; // Ok, let v16: unknown

let v17 = !v0; // Ok, let v17: boolean

Также стоит упомянуть, что функция, у которой возвращаемый тип принадлежит к типу unknown, может не возвращать значение явно.

function f0(): unknown {

return; // Ok

}

function f1(): number {

return; // Error

}

let v = f0(); // Ok, let v: unknown

При активной опции --strictPropertyInitialization принадлежащие к типу unknown поля не нуждаются в инициализации.

class T {

f0: unknown; // Ok

f1: number; // Error

f2: number = 5; // Ok

}

Если в определении типа данных участвует сопоставленный тип (Mapped Type), которому в качестве аргумента типа передается тип unknown, то такой сопоставленный тип будет выведен как объектный тип {}. Поскольку сопоставленные типы (Mapped Types), псевдонимы типов (types), а также обобщения (Generics<>) будут рассмотрены позднее, то стоит просто помнить об этом факте и повторно прочесть написанное при необходимости.

type MappedType<T> = {

[K in keyof T]: T;

};

type T0 = MappedType<number>; // type T0 = number

type T1 = MappedType<any>; // type T1 = { [x: string]: any; }

type T2 = MappedType<unknown>; // type T2 = {}

**Как вы отлавливаете ошибки в TypeScript коде?**

Functional programming languages such as TypeScript and JavaScript provide a way to handle errors using the **try-catch** blocks. The **try** block catches the errors, and the **catch** block handles the errors.  In this tutorial, we will learn to handle errors in TypeScript.

There are mainly 7 types of errors in TypeScript. Here, we will learn all types of errors one by one.

* **Range error** − If we try to access anything out of range, TypeScript will throw a range error. For example, accessing the array index for a large number value such as 10^100.
* **Reference error** − If we try to access the variable without declaring it, TypeScript throws a reference error. For example, without declaring the string variable, apply the toLowerCase() method on that variable.
* **Syntax error** − If we write the code with the wrong syntax, TypeScript throws a syntax error. For example, 10\*\*\*10 is a syntax error as \*\*\* is not representing any operator in TypeScript.
* **Type error** − It occurs due to variable type mismatch. For example, removing the decimal part of the string variable using the Math.ceil() method in TypeScript.
* **URI error** − It can occur while encoding or decoding the URI in TypeScript. For example, passing an invalid parameter to the decodeURI() function returns the URI error.
* **Eval error** − These errors occur while using the eval() function in TypeScript. We can use the eval() function to evaluate the expressions.
* **Internal error** − This kind of error occurs internally. For example, It can occur if stack size increases out of the limit.

We have looked at the different types of errors. Now, we will learn to handle the errors using the try, catch and finally block.

## Use the try, catch, and finally block

The first question arises − what is the try, catch, and finally block? The try block can contain the code with errors. If any error occurs, it throws the error and stops executing the code from the line where it occurred.

As the name suggests, the catch block is used to catch and handle errors. We can write the code to handle the errors in the catch block according to the different eros. There can be multiple catch blocks for a single try block. We can create a different catch block to handle different kinds of errors, such as syntax and reference errors.

The finally block contains the code that is always necessary to execute before program termination. For example, database connection closing or file closing, which we had opened in the try block before the error occurred.

### Syntax

Users can follow the syntax below to use the try, catch, and finally block.

try {

//  Code with error

} catch {

  // handle the error

} finally {

  // execute the code before the program termination

}

### Example 1

In the example below, we have demonstrated the use of the try-catch block to handle the errors in TypeScript. In the try block, we have thrown the error by creating a new instance of the error. The catch block catches and handles that error by printing some messages and errors on the screen. Also, users can see in the output that the “End of the try block” message is not printed as it is written in the line after the error occurred in the line.

try {

// This should generate syntax error.

throw new SyntaxError("Error occured inside the syntax.");

// this code will not execute.

console.log("End of the try block");

} catch (error) {

// control will be at here, after throwing the error from try block.

console.log("Inside the catch block.");

console.log("The error is " + error);

}

On compiling, it will generate the following JavaScript code −

try {

// This should generate syntax error.

throw new SyntaxError("Error occured inside the syntax.");

// this code will not execute.

console.log("End of the try block");

}

catch (error) {

// control will be at here, after throwing the error from try block.

console.log("Inside the catch block.");

console.log("The error is " + error);

}

### Output

The above code will produce the following output −

Inside the catch block.

The error is SyntaxError: Error occured inside the syntax.

### Example 2

In the example below, we have created the function inside the try block. It takes two parameters. The first parameter can be any value; another should be a number. Also, it always returns either number or object of the ReferenceError.

Inside the errofunc() function, we check that if both parameters are numbers, return the sum of the number; Otherwise, return the error. In the output, we can see that the first function call will execute properly, and the second function call will create an error.

try {

// Creating the function which returns an number or error

function errorfunc(num1, num2: number): number | ReferenceError {

// If both parameters of type number, then return addition.

// Otherwise throw reference error.

if (typeof num1 == "number" && typeof num2 == "number") {

console.log(num1 + num2);

}

// Throwing the new instance of the reference error.

throw new ReferenceError("Parameter/s are not type of numbers.");

}

// This will throw an error.

errorfunc("11", 12);

} catch (error) {

console.log("Inside the catch block.");

console.log("The error is " + error);

} finally {

// This will execute always

console.log(

"Inside the finally block to execute the code before program termination."

);

}

On compiling, it will generate the following JavaScript code −

try {

// Creating the function which returns an number or error

function errorfunc(num1, num2) {

// If both parameters of type number, then return addition.

// Otherwise throw reference error.

if (typeof num1 == "number" && typeof num2 == "number") {

console.log(num1 + num2);

}

// Throwing the new instance of the reference error.

throw new ReferenceError("Parameter/s are not type of number.");

}

// This will throw an error.

errorfunc("11", 12);

}

catch (error) {

console.log("Inside the catch block.");

console.log("The error is " + error);

}

finally {

// This will execute always

console.log("Inside the finally block to execute the code before program termination.");

}

### Output

The above code will produce the following output −

Inside the catch block.

The error is ReferenceError: Parameter/s are not type of number.

Inside the finally block to execute the code before program termination.

We learned to handle errors in this tutorial. We used the try, catch, and finally block to handle the errors. However, handling errors in TypeScript is similar to handling errors in JavaScript but we have to focus on the error types also in TypeScript.

<https://medium.com/geekculture/how-to-strongly-type-try-catch-blocks-in-typescript-4681aff406b9>

https://journal.plain.com/posts/2022-10-04-error-handling-in-typescript-like-a-pro/

**Type assertions**

Type assertion представляет модель преобразования значения переменной к определенному типу. Обычно в некоторых ситуациях одна переменная может представлять какой-то широкий тип, например, any или union, которые по факту допускают значения различных типов. Однако при этом нам надо использовать переменную как значение строго определенного типа. И в этом случае мы можем привести к этому типу.

В качестве примера возьмем простейшую задачу - на веб-странице есть html-элемент с id = header, и мы хотим получить этот элемент, например, чтобы изменить его текст. Для получения элемента по id мы могли бы использовать встроенную js-функцию document.getElementById():

const header = document.getElementById("header");

header.innerText = "Hello Typescript!";

Но в TypeScript эта функция возвращает объединение HTMLElement|null. То есть возвращаемое значение может представлять null, если соответствующий html-элемент отсутствует на веб-странице. Вследствие этого обращение к свойству header.innerText - к свойству объекта, который может быть null, содержит потенциальную ошибку. Поэтому компилятор при компиляции сгенерирует ошибку.

Однако ситуация может быть такова, что мы точно знаем, что у нас на странице есть такой элемент. Есть разные способы для решения этой проблемы, чтобы указать компилятору, что все нормально. И одним из ним является приведение типов с помощью type assertion.

Есть две формы приведения. Первая форма заключается в использовании угловых скобок:

const header = <HTMLElement>document.getElementById("header");

header.innerText = "Hello Typescript!";

Перед значением в угловых скобках указывается тип, к которому надо выполнить приведение. Так, в данном случае мы получаем объект типа HTMLElement

Вторая форма заключается в применении оператора as, после которого указывается тип, к которому надо выполнить преобразование:

const header = document.getElementById("header") as HTMLElement;

header.innerText = "Hello Typescript!";

Однако такие преобразования будут иметь силу, если мы точно знаем, что значение может быть преобразовано к целевому типу. Например, на странице есть элемент с id=header, поэтому мы можем преобразовать значение к типу HTMLElement. Однако если такого элемента нет, то во время выполнения мы опять же получим ошибку.