Дженерики TypeScript

в научно-художественном стиле



Дениев Халид

Дженерики TypeScript в научно-художественном стиле

Халид Дениев

Грозный, 2025

Содержание

Об авторе **4**

О книге **6**

Глава 1. Дейли **7**
1.1. Путешествие в TypeScript 9
1.2. Что такое дженерики? 9
1.3. Именование дженериков 11
1.4. Синтаксис дженериков 11
1.5. Озарение 14

Глава 2. Дженерики в функциях **15**
2.1. Явный вывод типа 17
2.2. Неявный вывод типа 18
2.3. const и неявный вывод типа 19
2.4. Несколько дженериков 19
2.5. Что нового мы узнали 20

Глава 3. Дженерики в типах **22**
3.1. Несколько дженериков 24

Глава 4. Дженерики в интерфейсах **26**
4.1. Код Степана 27
4.2. Комментарий Кати 27
4.3. Оператор extends 29
4.4. Значения по умолчанию 30

Глава 5. Дженерики в классах **32**
5.1. Оператор extends 35
5.2. Оператор keyof в методе класса 36

Глава 6. Операторы extends и keyof **40**
6.1. Пошаговый разбор 42

6.2. Оператор keyof 44
6.3. Оператор extends 44

Глава 7. Условные типы **47**
7.1. Условие для проверки на функцию 48
7.2. Условие для проверки на функцию 51

Глава 8. Дистрибутивность **53**
8.1. Как работает дистрибутивность 55
8.2. Как отключить дистрибутивность 56

Глава 9. Перегрузка функций **57**
9.1. Что такое перегрузка функций 58
9.2. Как пишут функцию с перегрузкой 60

Глава 10. Собеседование **61**

Глава 11. Мэппинг типов **65**
11.1. Как работает мэппинг 68

Заключение **70**

Об авторе

Привет, дорогой читатель! Меня зовут Дениев Халид. С программированием я подружился ещё в 2005-м, когда впервые попробовал PHP и JavaScript — с тех пор код стал моим главным увлечением.

За эти годы я прошёл путь от экспериментов с фреймворками и CMS на PHP (WordPress, Joomla, DLE, Kohana, CodeIgniter, Laravel, Yii) до создания собственных мини-фреймворков. Мне всегда важно понимать, что происходит «под капотом», поэтому я разбирал чужие решения и писал свои.

Я видел эволюцию веба с 2005 по 2025 годы: от первых скриптов и jQuery до React, Vue и Angular. Сейчас работаю как fullstack-разработчик: делаю мобильные приложения на React Native, пишу backend на Node.js с Express и Mongoose, создаю CLI-утилиты и настольные приложения на Electron.

За это время я поработал в 35 компаниях и успел поучаствовать в самых разных проектах. Теперь хочу делиться опытом в книгах — чтобы другим разработчикам было проще и быстрее проходить этот путь.

Если будут вопросы или идеи — пиши: **haliddeniev88@gmail.com**.

О книге

Технической литературы сегодня очень много. Но у меня часто складывается ощущение, что многие такие книги написаны не для людей. Сухие определения, формальные примеры, минимум живого языка — читать можно, понимать сложно.

В этой книге я решил пойти другим путём.

Это не строгий учебник. Скорее небольшой рассказ о том, как разработчик по имени Степан пытается разобраться в дженериках TypeScript. Он задаёт простые вопросы, иногда тупит, местами возмущается — как любой нормальный человек, когда сталкивается с неприветливым синтаксисом.

Я постарался объяснять тему не академически, а по-человечески. Там, где можно улыбнуться — мы улыбнёмся. Там, где нужно строго — будем строгими. Главное, чтобы после прочтения не осталось ощущения «сложно и страшно», а появилось «ясно и вроде даже интересно».

Если эта книга поможет тебе перестать бояться дженериков — значит, всё было написано не зря.

Глава 1. Дейли

Степан Кавычкин работает фронтенд-разработчиком в компании “React и точка”. Больше трёх лет он пишет код на JavaScript, пережил десятки дедлайнов, сотни ревью и тысячи коммитов с гордым сообщением “fix”.

На часах 09:59 — а значит, пора подключаться к дейли.

Степан открывает ноутбук, наливает кофе и подключается к звонку в “Яндекс Телемост”. На экране появляются знакомые лица: Петя с помятой головой и чаем, Маша на фоне идеальной белой стены, Дима с микрофоном, через который громче всего слышно кота.

В рабочем чате тестировщик Никита заранее предупредил: “Меня не будет, у меня запись к стоматологу”. Все дружно поставили смайлик с понимающей грустью, но в глубине души каждый знал: никакого стоматолога нет. Это тот самый мифический доктор, который появляется исключительно по утрам, когда душа категорически отказывается идти на дейли.

А вот у Димы была другая проблема — задачу он так и не сделал. И когда очередь дошла до него, вместо отчёта Дима включил актёрское мастерство:

— Меня слышно?… ой, похоже, со связью проблемы…

Он пару раз отключался, подключался, снова повторял *“меня слышно?”*, а потом внезапно пропал окончательно. Видимо, вместе с невыполненной задачей.

Наступает момент, когда слово передают Степану. Он кивает, делает вид, что бодр, и привычно произносит:

— Работаю над задачей 108, сегодня отправлю на ревью.

Эта фраза звучит почти как заклинание.

И тут тимлид, словно между прочим, добавляет:

— Кстати, Стёп, ребята пишут небольшой проект. Там всё на TypeScript. Отличная возможность подтянуть его, так что подключайся.

Степан делает вид, что всё в порядке, но внутри явно напрягся.

“TypeScript… ну вот, началось”, — думает он.

1.1. Путешествие в TypeScript

Сначала всё шло на удивление гладко. Степан быстро освоил типизацию параметров функций, научился описывать интерфейсы для объектов и даже понемногу разобрался с классами.

Редактор подсказывал ошибки заранее, и код стал аккуратнее. Степан подумал:

“Может, этот TypeScript не так уж страшен?”

Однако вскоре Степан столкнулся с тем, что выбило его из колеи.

Странные угловые скобки **<T>**, **<U>**, **<R>** встречались повсюду. Коллеги использовали их каждый день, документация была ими наполнена.

Степан пытался разобраться. Он открывал статьи, читал книги, смотрел примеры кода. Но чем больше он изучал, тем сильнее чувствовал:

“Я ничего не понимаю”.

И всё же после множества попыток он смог ухватить основное.

1.2. Что такое дженерики?

Дженерики — это механизм TypeScript, позволяющий создавать универсальные функции, интерфейсы, типы и классы, которые работают не с одним конкретным типом данных, а с абстрактным параметром.

Идея в том, что вместо жёсткой привязки к конкретному типу мы пишем общий шаблон, а нужный тип подставляется автоматически при использовании.

Таким образом, дженерики обеспечивают гибкость и переиспользуемость кода, сохраняя строгую типизацию и защиту от ошибок.

Если сказать по-научному, дженерики — это параметризованный полиморфизм.

Могу объяснить подробнее, если не против.

Представь: тебе нужна функция, которая принимает один аргумент и возвращает его же. Ты пишешь слово `function`, придумываешь имя функции, открываешь круглые скобки, вписываешь имя параметра, ставишь двоеточие и тип, закрываешь скобки. После скобок указываешь тип возвращаемого значения тем же способом, открываешь фигурные скобки, пишешь `return` и имя параметра, закрываешь фигурные:

```
function greeting(value: string): string {
    return value;
}
```

То есть дженерики — это тот же самый механизм объявления функции, только вместо того чтобы сразу ставить конкретные типы, ты после имени функции открываешь угловые скобки и вписываешь туда метки для будущих типов — например, `T`, `U`, `R`. Их может быть сколько угодно: иногда одна метка на всю функцию, иногда две или более.

Затем в круглых скобках ты подставляешь эти метки вместо конкретных типов — но не обязательно строго по порядку и не обязательно всем параметрам. Один параметр может использовать одну метку, два параметра могут использовать одну и ту же, а возвращаемое значение — третью.

А дальше, при вызове функции, TypeScript сам заменит твои метки на конкретные типы, исходя из того, что ты передашь в аргументы. Если хочешь, ты можешь передать типы вручную — тоже в угловых скобках.

1.3. Именование дженериков

Если ты впервые видишь `<T>` в коде, может показаться, что это какое-то особое слово из TypeScript. На самом деле — нет. `T` не зарезервировано, это просто имя параметра, которое выбрали по традиции.

Буква `T` — сокращение от слова `Type`. Ничто не мешает тебе назвать дженерик как угодно. TypeScript не обидится:

```
function foo<MyType>(value: MyType): MyType {
  return value;
}
```

Этот код полностью эквивалентен варианту с `T`. Здесь `MyType` — обычное имя, просто вместо `T` используется более понятное название.

Главное — чтобы имя было осмысленным, особенно если дженериков несколько.

> Не бойся отходить от буквы T, если контекст позволяет. В больших проектах осмысленные имена делают код гораздо понятнее. Через пару недель, открыв файл, ты сам себе скажешь спасибо.

1.4. Синтаксис дженериков

Дженерики записываются в угловых скобках `<T>` после имени функции, интерфейса, типа или класса. Принцип везде одинаковый: вместо конкретного типа указываем абстрактный параметр, который подставляется при использовании.

Далее мы подробно разберём синтаксис использования дженериков.

1.4.1. Синтаксис дженериков в функциях

Начнём с самого частого сценария — функций. У них есть несколько способов записи, и дженерики вписываются в каждый из них.

Обычная функция:

Это самый привычный синтаксис. После имени функции пишем <T>, а дальше используем T вместо конкретного типа.

```
function identity<T>(value: T): T {
  return value;
}
```

Функция-выражение:

Здесь всё работает так же, только функция не объявляется напрямую, а записывается в переменную. Дженерик указывается прямо перед параметрами:

```
const identity = function <T>(value: T): T {
  return value;
};
```

Стрелочная функция:

Современный вариант записи. Ничего не меняется — дженерик всё так же идёт перед аргументами:

```
const identity = <T>(value: T): T ⇒ {
  return value;
};
```

1.4.2. Синтаксис в типах

Дженерики можно использовать и в типах. Они позволяют создавать гибкие описания, которые подстраиваются под разные данные:

```
type Box<T> = {
  value: T;
};
```

Если в типе объявлён дженерик <T>, то при его использовании TypeScript потребует указать конкретный тип, иначе выдаст ошибку.

1.4.3. Синтаксис в интерфейсах

Аналогично дженерики работают и в интерфейсах:

```
interface Box<T> {
  value: T;
}
```

Как и в случае с типами, если в интерфейсе объявлён дженерик <T>, его нужно обязательно указать при использовании.

1.4.4. Синтаксис в классах

Дженерики можно использовать и в классах:

```
class Box<T> {
  private value: T;

  constructor(value: T) {
    this.value = value;
  }

  getValue(): T {
    return this.value;
  }

  setValue(newValue: T): void {
    this.value = newValue;
  }
}
```

Как и с типами и интерфейсами, дженерик в объявлении класса — это обязательный параметр.

1.5. Озарение

Степан облегчённо выдохнул: наконец-то в голове сложилось чёткое определение. Но едва появилась ясность, тут же накатила новая тревога:

“Хорошо, я понял, что такое дженерики. Но зачем они нужны на практике? Как их правильно применять? А вдруг я упускаю что-то важное?”

Он откинулся на спинку стула, потер глаза и уставился на телефон. Мысль пришла сама собой:

“Надо позвонить Николя…”