**TS + React = 💙**

Возможно, вы уже сталкивались с основной проблемой динамической типизации: структуры данных и их типы приходится постоянно держать в голове. А если вы что-то забыли, это в конечном итоге приводит к разным ошибкам. К примеру:

* не учли, что функция возвращает строку, а не число — получили «странный» результат вычисления в компоненте;
* не учли, что поле, получаемое с сервера, может быть пустым, — словили ошибку в рантайме.

Чем больше приложение, тем сложнее помнить обо всех сущностях, с которыми вы работаете, и всех тонкостях взаимодействия между модулями. А иногда приложения разрастаются до такой степени, что следить за типами и учитывать все нюансы становится попросту невыполнимой задачей (если только вы не Шелдон Купер).

Чтобы не пришлось запоминать метры кода и пить таблетки для памяти, в приложение можно добавить типизацию — пусть компьютер сам запоминает что и с чем ему следует складывать, а заодно и вам подсказывает.

TypeScript — одно из лидирующих решений для добавления типизации в проект. На его примере мы рассмотрим возможности типизации React-приложений.

Но для начала всё нужно установить и настроить. Об этом — в следующем уроке.

# Установка TypeScript

В этом уроке разберём варианты установки TypeScript — от создания нового приложения до внедрения TS в уже существующий проект.

## Создание приложения с поддержкой TS при помощи CRA

Как правило, новое React приложение удобно создавать c помощью утилиты create-react-app. Возможности этой утилиты можно расширять, применяя специальные шаблоны. Воспользуемся одним из них, чтобы создать новое приложения с поддержкой TS:

Скопировать кодBASH

npx create-react-app react-with-ts --template typescript

После выполнения этой команды будет создана директория react-with-ts. В ней развернётся уже знакомое вам NodeJS окружение для работы с React-приложениями, а также установятся все зависимости.

Чтобы проверить работоспособность проекта, достаточно перейти в директорию react-with-ts и выполнить команду npm run start (или yarn start).

У вас должен запуститься локальный сервер со стартовой страницей CRA:



Обратите внимание, что расширение файла App.tsx — .tsx. Оно используется для файлов, которые содержат JSX-синтаксис и прибегают к возможностям типизации TS.

Новое приложение с поддержкой TypeScript готово. Но что если у вас уже есть проект, в который вы решили добавить типизацию? Поговорим об этом далее.

## Добавление TS в существующее React приложение (CRA)

Стандартное create-react-app приложение уже частично подготовлено к работе с TypeScript, достаточно лишь установить необходимые зависимости:

Скопировать кодBASH

npm install --save typescript

npm install --save-dev @types/node @types/react @types/react-dom @types/jest

После этого вы можете создавать .tsx файлы и использовать в них TypeScript — Webpack автоматически определит эти файлы на этапе сборки и преобразует в JavaScript. При первой сборке в корне вашего проекта должен появиться файл tsconfig.json. Это конфигурационный файл TypeScript, в котором описываются правила синтаксиса и сборки.

Кстати, даже после добавления TS в проект, у вас останется возможность использовать обычный JS-синтаксис.

## Добавление TS в существующее React приложение (кастомный конфиг)

Установка TypeScript с настроенной вручную сборкой индивидуальна для каждого проекта: единого рецепта здесь нет. Всё зависит от того, какой сборщик вы используете, каким образом транспилируете JavaScript, какие библиотеки используете в проекте и так далее.

Но порядок действий примерно такой:

1. Устанавливаем сам TypeScript и набор типов для React:

Скопировать кодBASH

npm install --save typescript

npm install --save-dev @types/react @types/react-dom

1. Внедряем TypeScript в процесс сборки.
2. Создаём файл конфигурации tsconfig.json и описываем в нём правила сборки и синтаксиса TS.
3. Если в вашем приложении используются библиотеки, которые не поддерживают типизацию по умолчанию, нужно установить набор типов для этой библиотеки. Как правило, это npm пакет @types/название-библиотеки.

После этого ваше приложение будет готово к работе с TypeScript.

## Подготовка среды разработки

Обычно популярные редакторы текста и IDE способны распознавать TypeScript-синтаксис без дополнительных настроек. Но если вы столкнулись с проблемами — установите соответствующий TypeScript-плагин для вашей среды разработки.

Вот вы и узнали, как встроить TypeScript в проект. В следующем уроке настроим TS под свои нужды с помощью конфигурационного файла.

# Конфигурация TypeScript

TypeScript — очень гибкий инструмент. Его гибкость обеспечивают разные возможности конфигурации. Вы можете настроить параметры сборки, правила и уровень строгости типизации, определить конкретные директории и файлы, которые должны быть типизированы, и ещё много чего.

Конфигурация TypeScript описывается в файле tsconfig.json, который принято располагать в корне проекта.

Рассмотрим файл конфигурации, который генерируется при создании приложения с помощью утилиты create-react-app:

Скопировать кодJSON

// tsconfig.json

{

"compilerOptions": {

"target": "es5",

"lib": [

"dom",

"dom.iterable",

"esnext"

],

"allowJs": true,

"skipLibCheck": true,

"esModuleInterop": true,

"allowSyntheticDefaultImports": true,

"strict": true,

"forceConsistentCasingInFileNames": true,

"noFallthroughCasesInSwitch": true,

"module": "esnext",

"moduleResolution": "node",

"resolveJsonModule": true,

"isolatedModules": true,

"noEmit": true,

"jsx": "react-jsx"

},

"include": [

"src"

]

}

## Указание файлов

Начнём с простого: в поле include вы можете указать директории или специфический набор файлов, в которых будет применяться TypeScript. Все пути должны быть указаны относительно файла tsconfig.json.

Помимо include, существует поле exclude — в нём, наоборот, указываются директории или файлы, которые будут проигнорированы компилятором TypeScript.

## Настройки компилятора

Всё самое интересное происходит в поле compilerOptions — именно здесь описывается то, как TypeScript ведёт себя в проекте.

Вероятно, самое важное и полезное поле в этой конфигурации — strict. Оно влияет на режим строгой проверки типов, который не позволяет проводить операции между не соответствующими по типу объектами. Включение опции strict подталкивает разработчика правильно обращаться с типами, что приводит к повышению качества кода и устойчивости приложения.

Свойство target отвечает за результат компиляции TypeScript-кода. Так, при значении es5 код будет преобразован к стандарту ECMAScript 2009, который поддерживается подавляющим большинством браузеров.

Свойства module и moduleResolution позволяют настроить способ компиляции модулей. Благодаря этим настройкам вы можете свободно разбивать код приложения на модули, а компилятор соберёт этот код в JavaScript, понятный браузеру.

В свойстве lib указываются наборы определений типов, которые будут по умолчанию включены в проект. Например, приложение будет запускаться в браузере и вам нужна поддержка DOM API. Чтобы TypeScript мог распознать используемые вами методы, понадобится включить соответствующие определения типов — dom.

Важное для React свойство jsx отвечает за способ преобразования JSX-синтаксиса в JavaScript. Значение react-jsx говорит, что весь JSX код будет транспилирован, то есть преобразован, в вызов функции \_jsx из библиотеки react/jsx-runtime.

## Настройки окружения

Помимо основных настроек для компилятора, о том, как следует преобразовывать TypeScript-код, существует ещё множество настроек, которые влияют на поведение TS при разработке.

Например, параметр allowJs позволяет импортировать обычные .js файлы в TypeScript-проект. А параметр allowSyntheticDefaultImports упрощает синтаксис импорта модулей:

Скопировать кодTSX

*// allowSyntheticDefaultImports === true*

import React from "react";

*// allowSyntheticDefaultImports === false*

import \* as React from "react";

С полным списком настроек вы можете ознакомиться [в документации к TypeScript](https://www.typescriptlang.org/tsconfig).

В большинстве ситуаций вам не потребуется вручную заниматься конфигурацией TypeScript. Если для создания проекта вы используете CRA, файл tsconfig.json будет сгенерирован автоматически, а для собственных сборок можно воспользоваться [рекомендованной конфигурацией](https://github.com/tsconfig/bases#recommended-tsconfigjson). Но в случае, если вам понадобится настроить что-то под себя, теперь вы знаете, где и как это можно сделать.

С установкой и настройкой разобрались — TypeScript успешно прикручен к React. Теперь самое время понять, как применять эти технологии вместе.

# Типизация классовых компонентов

В предыдущей теме вы познакомились с синтаксисом TypeScript, теперь рассмотрим конкретные примеры использования TS с библиотекой React. Начнём с классовых компонентов.

## Типизация пропсов

Основной сценарий использования TypeScript в компонентах — типизация пропсов. Нужно быть уверенными в том, что компонент получит необходимый набор параметров и их использование не приведёт к ошибкам.

Рассмотрим компонент Movie:

Скопировать кодTYPESCRIPT

export class Movie extends Component {

render() {

const { title } = this.props;

return (

<div>

<h2>{title}</h2>

</div>

);

}

}

Этот компонент принимает один пропс title и рендерит его. Если во время разработки что-то пошло не так и в компонент забыли передать параметр title, контент в заголовке h2 окажется пустым, что можно считать ошибкой.

Исправить это можно с помощью типизации пропсов. Параметры классовых компонентов удобно описывать в интерфейсе, а типизировать компоненты с помощью дженериков:

Скопировать кодTYPESCRIPT

*// Интерфейс пропсов компонента Movie*

interface IMovieProps {

title: string;

}

*// Интерфайс добавляется в качестве дженерик-типа класса Component*

export class Movie extends Component<IMovieProps> {

render() {

const { title } = this.props;

return (

<div>

<h2>{title}</h2>

</div>

);

}

}

После этого внутри компонента Product можно смело использовать пропс title, так как TypeScript проследит за тем, чтобы этот параметр был передан компоненту:

Скопировать кодTYPESCRIPT

export default function App() {

return (

<div className="App">

{/\* Если забыть передать параметр, TypeScript покажет ошибку: \*/}

{/\* Property 'title' is missing in type '{}' but required in type 'Readonly<IProductsProps>' \*/}

<Movie />

{/\* TypeScript скомпилирует код только после передачи всех нужных параметров \*/}

<Movie title={'Карты, деньги, два ствола'} />

</div>

);

}

## Типизация children

Классы Component, PureComponent и FunctionComponent по умолчанию имеют типизацию пропса children. Например, если вы типизировали пропсы компонента с помощью интерфейса IMovieProps, итоговый тип пропсов будет выглядеть вот так: Readonly<IMovieProps> & Readonly<{ children?: ReactNode }>.

Но если вам понадобится явно указать children в списке пропсов, просто добавьте поле children в интерфейс компонента и определите его тип как ReactNode:

Скопировать кодTYPESCRIPT

interface IMovieProps {

title: string;

children?: ReactNode;

}

Тип ReactNode — комплексный тип, включающий множество элементов, которые может вернуть render() метод: type ReactNode = ReactChild | ReactFragment | ReactPortal | boolean | null | undefined;.

## Типизация параметров конструктора

Часто параметры необходимо использовать в конструкторе компонента. Если вы попробуете сделать всё как обычно, компилятор TS будет ругаться на неопределённый тип пропсов: Parameter 'props' implicitly has an 'any'. В этом случае компилятору нужно явно указать, какие параметры ожидаются в конструкторе:

Скопировать кодTYPESCRIPT

export class Movie extends Component<IMovieProps> {

constructor(props: IMovieProps) {

super(props);

*// ...*

}

*// ...*

}

## Типизация состояния компонента

Как вы уже знаете, помимо параметров, которые компонент получает извне, существует ещё и внутреннее состояние. Внутреннее состояние также можно типизировать. Этот процесс похож на тот, через который мы проходили при типизации пропсов: типы состояния описываются в отдельном интерфейсе и добавляются в компонент в качестве дженерика:

Скопировать кодTYPESCRIPT

*// ...*

interface IMovieState {

label: string;

}

*// Интерфайс добавляется в качестве дженерик-типа класса Component*

export class Movie extends Component<IMovieProps, IMovieState> {

*// ...*

}

После чего состояние нужно определить:

Скопировать кодTYPESCRIPT

interface IMovieState {

label: string;

}

export class Movie extends Component<IMovieProps, IMovieState> {

*// В state также необходимо типизировать*

state: Readonly<IMovieState> = {

label: "Художественный фильм"

};

render() {

const { title } = this.props;

const { label } = this.state;

return (

<div>

<h2>{title}</h2>

<span>{label}</span>

</div>

);

}

}

Обратите внимание на типизацию поля state: с помощью типа Readonly мы явно указываем компилятору, что данные в состоянии не могут быть изменены напрямую. Такой приём помогает избежать случайных ошибок, которые легко допустить в процессе разработки.

## Типизация методов жизненного цикла

Для типизации методов жизненного цикла достаточно указать типы параметров метода:

Скопировать кодTYPESCRIPT

export class Movie extends Component<IMovieProps, IMovieState> {

componentDidUpdate(prevProps: IMovieProps, prevState: IMovieState) {

*// ...*

}

}

В этом уроке мы разобрались, как типизировать классовые компоненты. Всё просто — достаточно добавить название типа в соответствующее место.

В следующем уроке рассмотрим возможности типизации функциональных компонентов.

type TButtonProps = {primary?: boolean, secondary?: boolean} & React.ButtonHTMLAttributes<HTMLButtonElement>;

export class Button extends PureComponent<TButtonProps> {

render() {

const { primary, secondary, children, ...props } = this.props;

return (

<button {...props} className={primary ? styles.primary : styles.secondary}>

{children}

</button>

);

}

}

В этом задании мы продолжим типизировать классовые компоненты React-приложения. В файле /components/password-input/index.tsx у компонента PasswordInput есть неточность: у него не типизирован стейт. Исправьте это с помощью дженерика.

import { Input, TInputProps } from '../input';

import eyeOff from '../../images/eye-off.svg';

import eye from '../../images/eye.svg';

import React, { FC } from 'react';

const EyeOff: FC<{ onClick: () => void }> = props => (

<img src={eyeOff} alt="eye-off" onClick={props.onClick} />

);

const Eye: FC<{ onClick: () => void }> = props => (

<img src={eye} alt="eye-off" onClick={props.onClick} />

);

export class PasswordInput extends React.Component<TInputProps, {isVisible: boolean}> {

state: Readonly<{isVisible: boolean}> = { isVisible: false };

render() {

const { type, ...props } = this.props;

return (

<Input

{...props}

type={this.state.isVisible ? 'text' : 'password'}

icon={this.state.isVisible ? EyeOff : Eye}

onIconClick={() => this.setState({ isVisible: !this.state.isVisible })}

/>

);

}

}

import styles from './input.module.css';

import React, { FC } from 'react';

export type TInputProps = {

onIconClick?: () => void;

icon?: React.ElementType;

} & React.InputHTMLAttributes<HTMLInputElement>;

export const Input: FC<TInputProps> = ({

icon: Icon,

onIconClick,

value,

placeholder,

onChange,

type,

...props

}) => {

const icon = Icon ? <Icon onClick={onIconClick} className={styles.inputContainer} /> : null;

return (

<div className={styles.inputContainer}>

<input

className={styles.input}

type={type}

value={value}

placeholder={placeholder}

onChange={onChange}

{...props}

/>

{icon}

</div>

);

};

# Типизация функциональных компонентов

В этом уроке разберём, как типизировать параметры в функциональном компоненте. Для этого существует несколько методов, которые можно применять в зависимости от ситуации.

Самый простой способ — типизировать пропсы как аргумент функции:

Скопировать кодTYPESCRIPT

interface IMovieProps {

title: string;

}

export const Movie = ({ title }: IMovieProps) => {

return (

<div>

<h2>{title}</h2>

</div>

);

};

В этом случае будет работать почти всё, но есть один момент: если вы попробуете передать компоненту какой-либо children, TypeScript не даст вам этого сделать и будет ругаться на неизвестный параметр: Property 'children' does not exist on type 'IntrinsicAttributes & IMovieProps'. Всё логично, ведь в интерфейсе IMovieProps действительно отсутствует поле children.

Так что этот вариант типизации вряд ли нас устроит, ведь обычно внутрь одних компонентов необходимо передавать другие. На первый взгляд эту проблему можно решить, если определить children в интерфейсе IMovieProps, но и это не самый удобный способ. Гораздо лучше воспользоваться заранее созданными типами библиотеки React.

Для этого определим тип самого компонента с помощью FunctionComponent:

Скопировать кодTYPESCRIPT

export const Movie: FunctionComponent = () => {

*// ...*

}

А для типизации пропсов передадим их тип в качестве дженерика:

Скопировать кодTYPESCRIPT

interface IMovieProps {

title: string;

}

export const Movie: FunctionComponent<IMovieProps> = ({ title }) => {

return (

<div>

<h2>{title}</h2>

</div>

);

};

Теперь компонент полностью типизирован и будет работать без проблем.

Кстати, у типа FunctionComponent есть более короткая запись, которую удобно применять на практике — FC. Например, компонент Movie можно типизировать так:

Скопировать кодTYPESCRIPT

interface IMovieProps {

title: string;

}

export const Movie: FC<IMovieProps> = ({ title }) => {

return (

<div>

<h2>{title}</h2>

</div>

);

};

FC — не отдельный тип компонентов, а лишь псевдоним типа FunctionComponent:

type FC<P = {}> = FunctionComponent<P>

## Необязательные пропсы

У компонента могут быть как обязательные параметры, так и необязательные. Чтобы определить тип необязательного параметра, достаточно объявить его в интерфейсе:

Скопировать кодTYPESCRIPT

interface IMovieProps {

title: string;

label?: string;

}

export const Movie: FunctionComponent<IMovieProps> = ({ title, label }) => {

return (

<div>

<h2>{title}</h2>

{label && <span>{label}</span>}

</div>

);

};

После этого TypeScript будет знать о том, что тип поля label может быть как string, так и undefined, и подскажет вам, что эту особенность нужно учесть при описании логики компонента.

Вот и всё. Вы узнали, как типизировать параметры в функциональном компоненте — это оказалось не сложнее, чем типизация классовых компонентов.

В следующем уроке разберём возможности типизации хуков.

import React, { FC } from 'react';

import styles from './button.module.css';

type TButtonProps = {

primary?: boolean;

secondary?: boolean;

} & React.ButtonHTMLAttributes<HTMLButtonElement>;

export const Button: FC<TButtonProps> = ({ primary, secondary, ...props }, children) => {

return (

<button {...props} className={primary ? styles.primary : styles.secondary}>

{children}

</button>

);

}

# Типизация хуков

У хуков в библиотеке React достаточно специфичный синтаксис, и бывает не просто разобраться, какими сущностями они оперируют. Эту задачу можно облегчить с помощью типизации — вы будете уверены в том, что не допустили ошибки при написании параметров и все хуки отработают корректно.

В этом уроке разберём, как типизировать хуки, которые чаще всего встречаются на практике.

## useState

По умолчанию тип этого хука задаётся на основе начального значения:

Скопировать кодTYPESCRIPT

*// В этом случае тип price будет автоматически определён как number*

const [price, setPrice] = useState(300);

Если вам необходимо явно определить тип состояния, можно воспользоваться дженериком:

Скопировать кодTYPESCRIPT

const [price, setPrice] = useState<number>(300);

Но бывают ситуации, когда начальное значение неизвестно и хуку useState нечего передать в качестве аргумента. Чтобы в этой ситуации ничего не сломалось, нужно явно указать тип состояния:

Скопировать кодTYPESCRIPT

const [price, setPrice] = useState<number>();

В таком случае итоговой тип состояния price определяется на основе дженерика и, так как начального значения не существует, дополняется типом undefined. В итоге получается такой тип: number | undefined.

А полная запись типа возвращаемого значения хука будет выглядеть так: [number | undefined, Dispatch<SetStateAction<number | undefined>>].

Если в стейте нужно хранить сложный тип данных, используется такой же метод типизации, как и в случае с примитивами:

Скопировать кодTYPESCRIPT

interface IUser {

name: string;

age: number;

}

const [user, setUser] = useState<IUser>();

## useReducer

Для типизации хука useReducer достаточно определить типы параметров state и action:

Скопировать кодTYPESCRIPT

*// Интерфейс состояния*

interface IState {

count: number;

}

*// Интерфейс события*

interface IAction {

type: 'increment' | 'decrement';

}

const initialState: IState = { count: 0 };

*// Типы присваиваются в функции-редьюсере*

function reducer(state: IState, action: IAction) {

switch (action.type) {

*// ...*

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, initialState);

*// ...*

}

## useContext

Функция createContext умеет определять тип по начальному значению, что удобно при создании контекста, в котором хранятся примитивы. Но, как правило, в контексте хранятся сложные типы данных. В таком случае тип добавляется с помощью дженерика:

Скопировать кодTYPESCRIPT

type TTheme = 'light' | 'dark';

const ThemeContext = createContext<TTheme>('dark');

*// В таком случае тип ThemeContext будет определён как Context<TTheme>*

Хук useContext наследует типизацию от переданного в него контекста:

Скопировать кодTYPESCRIPT

const theme = useContext(ThemeContext);

## useEffect

Хук useEffect используется для выполнения сайд-эффектов и не возвращает никаких значений. Поэтому типизировать его не нужно, но мы всё же разберёмся с его интерфейсом — для лучшего понимания:

function useEffect(effect: EffectCallback, deps?: DependencyList): void;

Хук useEffect может принимать два аргумента:

* effect — функция с типом EffectCallback. Эта функция либо ничего не возвращает, либо возвращает функцию-деструктор (функцию-сброс).
* deps — список зависимостей хука, DependencyList. Представляет собой массив, доступный только для чтения, то есть ReadonlyArray. Так как это необязательный аргумент, запись его типа может выглядит так: DependencyList | undefined.

Для примера разберём компонент Timer:

Скопировать кодTYPESCRIPT

interface ITimerProps {

tick: number;

}

export const Timer: FunctionComponent<ITimerProps> = ({ tick }) => {

const [value, setValue] = useState<number>(0);

useEffect(

*// первый аргумент — функция EffectCallback*

() => {

const timerInterval = setInterval(() => {

setValue((value) => value + 1);

}, tick);

*// возвращается функция Destructor*

return () => {

clearInterval(timerInterval);

};

},

*// второй аргумент — список зависимостей DependencyList*

[tick]

);

return (<span>{value}</span>);

};

## useRef

Хук useRef также старается определить тип по начальному значению:

Скопировать кодTYPESCRIPT

const ref = useRef(null);

*// Тип такого рефа будет автоматически определён как:*

MutableRefObject<null>

Но если применять useRef для создания ссылок на DOM-элемент, это поведение нам совсем не подходит, так как невозможно положить HTMLElement в хранилище с типом null.

В этом случае необходимо явно задать тип хранимого элемента с помощью дженерика:

Скопировать кодTYPESCRIPT

const inputRef = useRef<HTMLInputElement>(null);

*// В этом случае тип рефа будет:*

RefObject<HTMLInputElement>

Теперь можно без проблем вкладывать нужный элемент в реф:

Скопировать кодTYPESCRIPT

export const ComponentWithRef: FunctionComponent = () => {

const inputRef = useRef<HTMLInputElement>(null);

const handleButtonClick = () => {

if(inputRef && inputRef.current) {

inputRef.current.focus();

}

};

return (

<div>

<input type="text" ref={inputRef}/>

<button onClick={handleButtonClick}>

Focus

</button>

</div>

);

};

## useMemo

Тип результата хука useMemo определяется автоматически на основании значения, которое возвращает функция:

Скопировать кодTYPESCRIPT

interface IExponentProps {

base: number;

exponent: number;

}

export const Exponent: FunctionComponent<IExponentProps> = ({base, exponent}) => {

*// В этом случае тип value будет number*

const value = useMemo(() => Math.pow(base, exponent), [base, exponent]);

return (<div>{value}</div>);

};

Вторым аргументом useMemo ожидает DependencyList | undefined — то же самое, что и хук useEffect.

Кроме автоматического определения типа, можно задать тип возвращаемого значения вручную с помощью дженерика:

Скопировать кодTYPESCRIPT

const value = useMemo<string>(() => 'Hi there!', []);

## useCallback

Хук useCallback работает аналогично useMemo. Но если вы решите задать тип с применением дженерика, следует определить тип всей функции, а не только возвращаемого значения:

Скопировать кодTYPESCRIPT

type exponentCallback = (base: number) => number;

const value = useCallback<exponentCallback>((base) => Math.pow(base, exponent), [exponent])

Это всё, что нужно знать о типизации хуков. Теперь самое время закрепить материал на практике.

# Где искать типизацию? А если не нашлась?

Web-разработчики не только пишут собственный код, но и постоянно используют разные библиотеки, которые уже написаны на JavaScript. При добавлении такой библиотеки в TypeScript проект, велика вероятность столкнуться с подобной ошибкой:

Could not find a declaration file for module 'library-name'.

Дело в том, что для каждого импортируемого JS модуля нужно создать специальный файл, в котором описываются типы всех значений, экспортируемых из этого модуля.

Хорошая новость: для подавляющего большинства библиотек такие файлы уже созданы, а в этом уроке вы узнаете, где их искать и как устанавливать.

## Как устанавливать файлы деклараций

Для удобства сообщество разработчиков создало [репозиторий ”Definitely Typed“](https://github.com/DefinitelyTyped/DefinitelyTyped), в котором хранится большинство файлов деклараций для различных JavaScript библиотек. Фишка репозитория в том, что он связан с NPM скоупом (англ. scope) @types, благодаря чему создаётся единый интерфейс установки типов для любой библиотеки.

Итак, чтобы добавить в проект файлы декларации любой библиотеки, вам необходимо воспользоваться командой:

Скопировать кодBASH

npm install --save-dev @types/library-name

После чего TypeScript «признает» JavaScript библиотеку и сможет с ней работать.

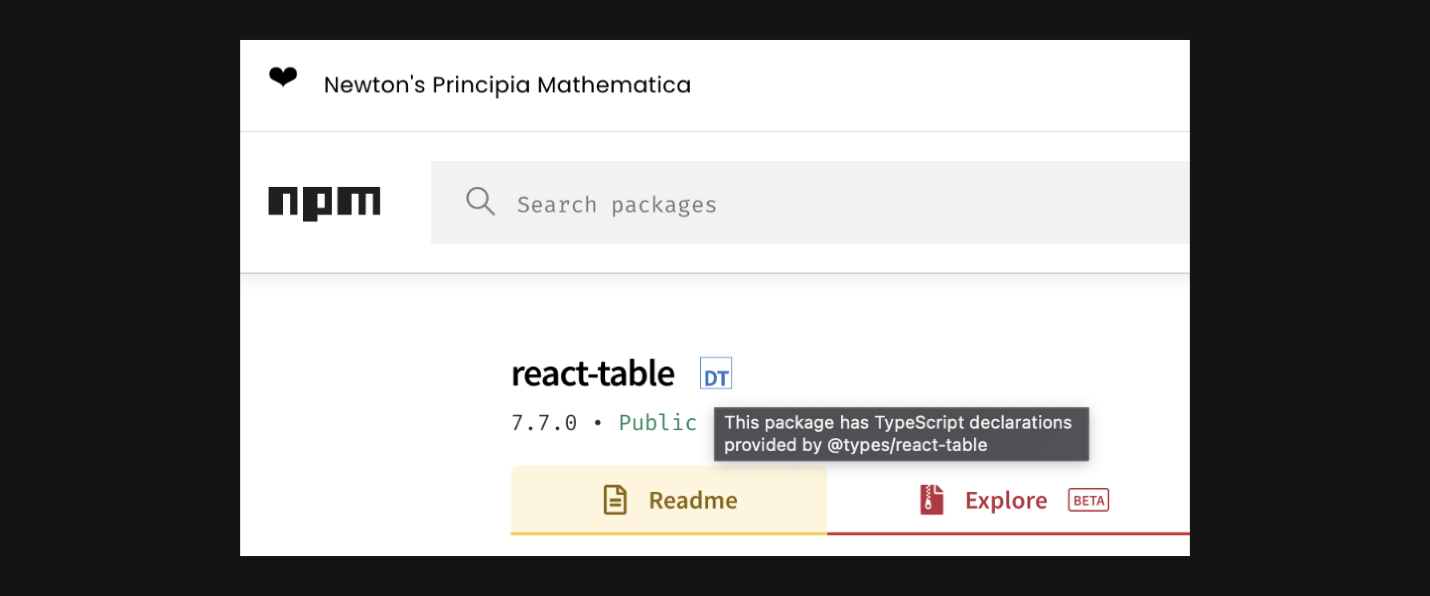
Кстати, [в начале этой темы](https://praktikum.yandex.ru/trainer/react/lesson/a28d90e1-31ed-47e5-b5e5-a08e81ef670b) вы уже устанавливали файлы деклараций для библиотеки React:

Скопировать кодBASH

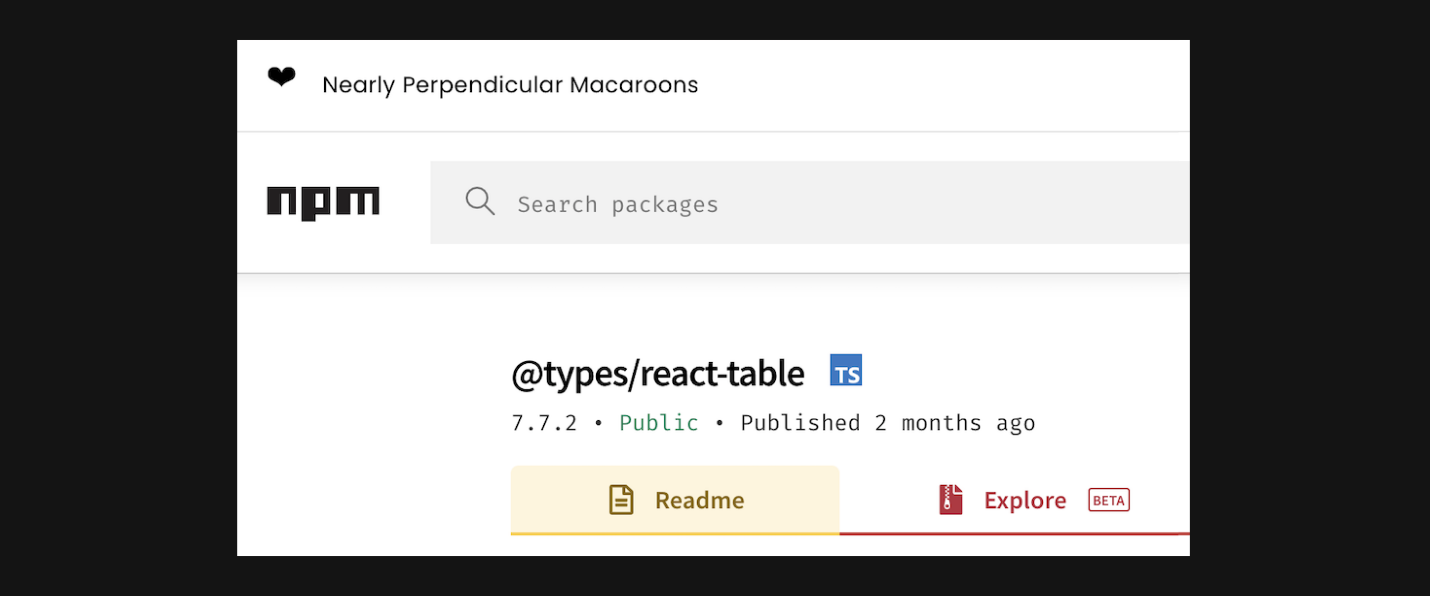
npm install --save-dev @types/react @types/react-dom

### Подсказки от npmjs.com

Когда вы ищите npm-пакет, в реестре [npmjs.com](http://npmjs.com/) можно увидеть отметки DT и TS рядом с названием пакета. Например, react-table:



DT — ссылка, которая ведёт на файлы декларации для конкретной библиотеки. По ссылке можно найти @types/react-table, но уже с отметкой TS, которая показывает принадлежность к репозиторию ”Definitely Typed“:



## Что делать, если типов в @types нет

Бывают ситуации, когда типов для библиотеки нет, а использовать её очень хочется. В таком случае можно типизировать модуль вручную.

Для этого нужно создать в проекте файл library-name.d.ts, сохранить который можно в любом месте, например, в директории declarations в корне проекта.

Чтобы TypeScript смог найти ваши файлы с типами, необходимо в файле конфигурации tsconfig.json в поле include указать путь до директории declarations:

Скопировать кодJSON

"include": [

"./src",

"./declarations"

],

В файле library-name.d.ts вам нужно описать интерфейсы методов, которые вы собираетесь использовать в проекте.

Например, если библиотека экспортирует такую функцию:

Скопировать кодJSX

export function sayHi(name) {

console.log(`Hi ${name}`);

}

То в файле library-name.d.ts следует добавить:

Скопировать кодTYPESCRIPT

declare function sayHi(name: string): void;

После чего TypeScript сможет распознать типы этой функции, и её можно будет свободно использовать в проекте.

Но такой способ применения файлов декларации очень редок. Гораздо чаще разработчики сталкиваются с проблемой импортирования файлов, которые не являются модулями.

Вероятно, самый распространённый пример такого импорта — css-modules. Дело в том, что в .сss просто нет знакомых компилятору TypeScript структур данных. Поэтому вам нужно научить TS работать с такими файлами. Сделать это можно, определив в файле декларации интерфейс работы с css-modules:

Скопировать кодTYPESCRIPT

*// css-modules.d.ts*

declare module '\*.css' {

interface IClassNames {

[className: string]: string;

}

const classNames: IClassNames;

export = classNames;

}

Таким образом вы сообщаете компилятору, что при импорте из \*.css файла возвращается объект classNames с набором полей, в которых лежат строковые значения.

Вот мы и разобрали, что такое файлы деклараций, где их искать и как создавать самим. Теперь ничто не помешает вам использовать в проекте с TypeScript любые JavaScript (и не только) файлы.