

основы

TypeScript

Адам Фримен



MANNING





Essential TypeScript 5

THIRD EDITION

ADAM FREEMAN



MANNING
SHELTER ISLAND

Основы TypeScript

АДАМ ФРИМЕН



Санкт-Петербург · Москва · Минск

2024

ББК 32.988.02-018
УДК 004.738.5

Фримен Адам

Ф88 Основы TypeScript. — СПб.: Питер, 2024. — 576 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2215-8

TypeScript — популярная надстройка над JavaScript с поддержкой статической типизации, которая наверняка покажется знакомой программистам на C# или Java. TypeScript поможет вам сократить количество ошибок и повысить общее качество кода на JavaScript.

«Основы TypeScript» — это полностью обновленное третье издание классического бестселлера Адама Фримена. В нем освещены все возможности TypeScript 5, включая новые, такие как декораторы. Сначала вы узнаете, зачем и почему был создан язык TypeScript, а затем почти сразу перейдете к практическому применению статических типов. Ничего лишнего! Каждая глава посвящена навыкам, необходимым для написания потрясающих веб-приложений.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

Права на издание получены по соглашению с Manning Publications Co. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав. Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

Издательство не несет ответственности за достоверность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1633437319 англ.

© Authorized translation of the English edition © 2023 Manning Publications.
This translation is published and sold by permission of Manning Publications,
the owner of all rights to publish and sell the same.
© Перевод на русский язык ООО «Прогресс книга», 2024
© Издание на русском языке, оформление ООО «Прогресс книга», 2024
© Серия «Библиотека программиста», 2024

ISBN 978-5-4461-2215-8

Краткое содержание

Предисловие	20
О книге	21
Об авторе	23
О научном редакторе	24
Иллюстрация на обложке	25
От издательства	26
Глава 1. Общая информация	27

Часть I. Начало работы с TypeScript

Глава 2. Ваше первое приложение на TypeScript	36
Глава 3. Обзор JavaScript, часть 1	69
Глава 4. Обзор JavaScript, часть 2	104
Глава 5. Компилятор TypeScript	135
Глава 6. Тестирование и отладка TypeScript	162

Часть II. Возможности TypeScript

Глава 7. Статические типы	180
Глава 8. Функции	208

6 Краткое содержание

Глава 9. Массивы, кортежи и перечисления	227
Глава 10. Объекты.	255
Глава 11. Работа с классами и интерфейсами.	282
Глава 12. Обобщенные типы	320
Глава 13. Расширенные возможности обобщенных типов.	350
Глава 14. Декораторы.	384
Глава 15. Работа с JavaScript	417

Часть III. Создание веб-приложений на основе TypeScript

Глава 16. Создание автономного веб-приложения, часть 1	442
Глава 17. Создание автономного веб-приложения, часть 2	472
Глава 18. Создание приложения на Angular, часть 1	492
Глава 19. Создание приложения на Angular, часть 2	514
Глава 20. Создание приложения на React, часть 1	529
Глава 21. Создание приложения на React, часть 2	553

Оглавление

Предисловие	20
О книге.	21
Кому стоит прочитать эту книгу	21
Структура издания	21
О коде.	22
Об авторе	23
О научном редакторе	24
Иллюстрация на обложке	25
От издательства	26
Глава 1. Общая информация	27
1.1. Стоит ли использовать TypeScript	27
1.1.1. Чем полезен TypeScript для повышения производительности разработчиков	28
1.1.2. Особенности версий JavaScript	29
1.2. Что нужно знать	30
1.3. Как настроить среду разработки	30
1.4. Какова структура этой книги	30
1.5. Много ли примеров?	31
1.6. Где можно получить код примера	32
1.7. Что делать, если у вас возникли проблемы с выполнением примеров.	33
1.7.1. Что делать, если вы обнаружили ошибку в книге	33

8 Оглавление

1.8. Как связаться с автором	33
1.9. Если вам понравилась эта книга	34
1.10. Если вам не понравилась эта книга	34
Резюме	34

Часть I. Начало работы с TypeScript

Глава 2. Ваше первое приложение на TypeScript	36
2.1. Первые шаги.	37
2.1.1. Установка Node.js	37
2.1.2. Установка Git	37
2.1.3. Установка TypeScript	37
2.1.4. Установка текстового редактора кода	38
2.2. Создание проекта	39
2.2.1. Инициализация проекта	39
2.2.2. Создание файла конфигурации компилятора	39
2.2.3. Добавление файла с кодом TypeScript	40
2.2.4. Компиляция и выполнение кода	40
2.2.5. Определение модели данных.	41
2.2.6. Добавление функций в класс коллекции	46
2.3. Использование пакетов сторонних производителей.	53
2.3.1. Подготовка к работе со сторонним пакетом	53
2.3.2. Установка и использование сторонних пакетов	55
2.3.3. Добавление деклараций типов для пакета JavaScript	57
2.4. Добавление команд	59
2.4.1. Фильтрация элементов	59
2.4.2. Добавление задач	60
2.4.3. Отметка о выполнении задачи	62
2.5. Постоянное хранение данных	65
Резюме	68

Глава 3. Обзор JavaScript, часть 1	69
3.1. Первоначальные приготовления	69
3.2. Запутанный JavaScript	71
3.3. Типы данных в JavaScript	72
3.3.1. Работа с примитивными типами данных	73

3.3.2. Приведение типа	75
3.3.3. Работа с функциями	79
3.4. Работа с массивами	84
3.4.1. Оператор расширения spread для массивов	86
3.4.2. Деструктуризация массивов	86
3.5. Работа с объектами	88
3.5.1. Добавление, изменение и удаление свойств объекта	88
3.5.2. Использование операторов spread и rest для объектов	91
3.5.3. Определение геттеров и сеттеров	93
3.5.4. Определение методов	94
3.6. Ключевое слово this	96
3.6.1. Ключевое слово this в автономных функциях	97
3.6.2. Ключевое слово this в методах	98
3.6.3. Изменение поведения ключевого слова this	99
3.6.4. Ключевое слово this в стрелочных функциях	100
3.6.5. Возвращаясь к исходной проблеме	101
Резюме	103
Глава 4. Обзор JavaScript, часть 2	104
4.1. Первоначальные приготовления	104
4.2. Наследование объектов в JavaScript	105
4.2.1. Проверка и изменение прототипа объекта	106
4.2.2. Пользовательские прототипы	108
4.2.3. Функции-конструкторы	109
4.2.4. Цепочка функций-конструкторов	110
4.2.5. Проверка типов прототипов	113
4.2.6. Определение статических свойств и методов	114
4.2.7. Классы в JavaScript	114
4.3. Итераторы и генераторы	119
4.3.1. Использование генератора	120
4.3.2. Определение итерируемых объектов	121
4.4. Коллекции в JavaScript	124
4.4.1. Хранение данных по ключу с использованием объекта	124
4.4.2. Хранение данных по ключу с использованием тар	125
4.4.3. Использование символов для ключей тар	126
4.4.4. Хранение данных по индексу	127

10 Оглавление

4.5. Модули	128
4.5.1. Объявление типа модуля	129
4.5.2. Создание модуля JavaScript	129
4.5.3. Использование модуля JavaScript	130
4.5.4. Экспорт именованных функций из модуля.	131
4.5.5. Определение нескольких именованных функций в модуле	133
Резюме	134
Глава 5. Компилятор TypeScript	135
5.1. Первоначальные приготовления	135
5.2. Структура проекта	137
5.3. Менеджер пакетов Node	138
5.4. Файл конфигурации компилятора.	141
5.5. Компиляция TypeScript-кода	143
5.5.1. Об ошибках компилятора.	144
5.5.2. Режим отслеживания и выполнение скомпилированного кода	145
5.6. Функция выбора целевой версии	148
5.7. Установка файлов библиотек для компиляции.	150
5.8. Выбор формата модуля	152
5.8.1. Указание формата модуля	155
5.9. Полезные параметры конфигурации компилятора	158
Резюме	160
Глава 6. Тестирование и отладка TypeScript	162
6.1. Первоначальные приготовления	162
6.2. Отладка TypeScript-кода	163
6.2.1. Подготовка к отладке	163
6.2.2. Использование Visual Studio Code для отладки.	164
6.2.3. Использование встроенного отладчика Node.js	166
6.2.4. Функция удаленной отладки Node.js.	167
6.3. Линтер TypeScript	169
6.3.1. Отключение правил линтера.	170
6.4. Модульное тестирование TypeScript	173
6.4.1. Настройка тестового фреймворка.	174
6.4.2. Создание модульных тестов	175
6.4.3. Запуск тестового фреймворка	176
Резюме	178

Часть II. Возможности TypeScript

Глава 7. Статические типы	180
7.1. Первоначальные приготовления	182
7.2. Статические типы	183
7.2.1. Создание статического типа с помощью аннотации типа	185
7.2.2. Использование неявно определенных статических типов	186
7.2.3. Тип any	189
7.3. Объединения типов	192
7.4. Утверждения типа	194
7.4.1. Утверждение к неожиданному типу	195
7.5. Защита типа	196
7.5.1. Тип never	198
7.6. Тип unknown	198
7.7. Тип NULL	200
7.7.1. Ограничение присвоений значения null	201
7.7.2. Удаление null из объединения с помощью утверждения	202
7.7.3. Удаление null из объединения с помощью защиты типа	204
7.7.4. Утверждение определения присваивания	204
Резюме	206
Глава 8. Функции	208
8.1. Первоначальные приготовления	210
8.2. Определение функций	211
8.2.1. Переопределение функций	211
8.2.2. Параметры функций	212
8.2.3. Результаты функции	219
8.2.4. Перегрузка типов функций	222
8.2.5. Функции утверждения	223
Резюме	225
Глава 9. Массивы, кортежи и перечисления	227
9.1. Первоначальные приготовления	228
9.2. Работа с массивами	230
9.2.1. Вывод типов массивов	231
9.2.2. Проблемы при выводе типов массивов	232
9.2.3. Проблемы с пустыми массивами	233

12 Оглавление

9.3. Кортежи	234
9.3.1. Обработка кортежей	235
9.3.2. Использование типов кортежей	237
9.3.3. Кортежи с необязательными элементами	238
9.3.4. Определение кортежей с rest-элементами	239
9.4. Перечисления	239
9.4.1. Принцип работы перечислений	241
9.4.2. Строковые перечисления	244
9.4.3. Ограничения перечислений	244
9.5. Типы с литеральным значением	247
9.5.1. Типы с литеральным значением в функциях	249
9.5.2. Смешивание типов значений в типе с литеральным значением	249
9.5.3. Переопределения типов с литеральными значениями	250
9.5.4. Шаблонный литеральный строковый тип	251
9.6. Псевдонимы типов	252
Резюме	254
Глава 10. Объекты	255
10.1. Первоначальные приготовления	256
10.2. Работа с объектами	257
10.2.1. Использование аннотаций структуры типа объекта	259
10.2.2. О соответствии структур типов	259
10.2.3. Использование псевдонимов типов для структур типов	263
10.2.4. Объединения структур типов	264
10.2.5. Объединение типов свойств	265
10.2.6. Защита типов для объектов	266
10.3. Пересечения типов	270
10.3.1. Использование пересечений для сопоставления данных	272
10.3.2. Объединение пересечений	273
Резюме	280
Глава 11. Работа с классами и интерфейсами	282
11.1. Подготовка к изучению главы	283
11.2. Функции-конструкторы	285
11.3. Классы	287
11.3.1. Управление доступом с помощью ключевых слов	289
11.3.2. Приватные поля в JavaScript	291

11.3.3. Определение свойств, доступных только для чтения	292
11.3.4. Упрощение конструкторов классов	293
11.3.5. Определение аксессоров	294
11.3.6. Автоаксессоры	299
11.3.7. Наследование классов	300
11.3.8. Абстрактные классы.	303
11.4. Использование интерфейсов.	306
11.4.1. Реализация нескольких интерфейсов.	308
11.4.2. Расширение интерфейсов	310
11.4.3. Необязательные свойства и методы интерфейса	311
11.4.4. Определение реализации абстрактного интерфейса.	313
11.4.5. Защита типа интерфейса	314
11.5. Динамическое создание свойств	315
11.5.1. Включение проверки значения индекса	316
Резюме	319
Глава 12. Обобщенные типы	320
12.1. Первоначальные приготовления	321
12.2. Проблемы, решаемые обобщенными типами	323
12.2.1. Добавление поддержки другого типа	324
12.3. Создание обобщенных классов	325
12.3.1. Аргументы обобщенных типов	327
12.3.2. Использование аргументов различных типов.	328
12.3.3. Ограничение значений обобщенных типов	329
12.3.4. Определение нескольких параметров типа	332
12.3.5. Разрешение компилятору выводить аргументы типа	334
12.3.6. Расширение обобщенных классов	336
12.3.7. Защита обобщенных типов	340
12.3.8. Определение статического метода в обобщенном классе.	342
12.4. Определение обобщенных интерфейсов	344
12.4.1. Расширение обобщенных интерфейсов.	345
12.4.2. Реализация обобщенного интерфейса	345
Резюме	349
Глава 13. Расширенные возможности обобщенных типов.	350
13.1. Первоначальные приготовления	351
13.2. Использование обобщенных коллекций	352

13.3. Использование обобщенных итераторов	354
13.3.1. Объединение итерируемого и итератора	356
13.3.2. Создание итерируемого класса	357
13.4. Индексные типы	358
13.4.1. Запрос индексного типа	358
13.4.2. Явное указание параметров обобщенных типов для индексных типов	359
13.4.3. Оператор индексированного доступа	360
13.4.4. Использование индексного типа для класса collection<t>	362
13.5. Сопоставление типа	364
13.5.1. Изменение имен и типов сопоставления	365
13.5.2. Параметр обобщенного типа с сопоставленным типом	366
13.5.3. Изменение обязательности и изменчивости свойств	367
13.5.4. Основные встроенные сопоставления	368
13.5.5. Комбинирование преобразований в одном сопоставлении	370
13.5.6. Создание типов с помощью сопоставления типов	371
13.6. Условные типы	372
13.6.1. Вложенные условные типы	373
13.6.2. Условные типы в обобщенных классах	374
13.6.3. Использование условных типов в объединениях типов	375
13.6.4. Использование условных типов в сопоставлениях типов	377
13.6.5. Определение свойств конкретного типа	378
13.6.6. Вывод дополнительных типов в условиях	379
Резюме	383
Глава 14. Декораторы.	384
14.1. Первоначальные приготовления	385
14.2. Что такое декораторы	387
14.2.1. Использование контекстных данных декоратора	390
14.2.2. Использование определенных типов в декораторе.	393
14.3. Другие типы декораторов.	395
14.3.1. Декоратор класса.	395
14.3.2. Декораторы полей	398
14.3.3. Декораторы аксессоров.	400
14.3.4. Декораторы автоаксессоров	403

14.4. Передача дополнительного аргумента декоратору	406
14.5. Применение нескольких декораторов	410
14.6. Инициализаторы	412
14.7. Накопление данных о состоянии	414
Резюме	416
Глава 15. Работа с JavaScript	417
15.1. Первоначальные приготовления	418
15.1.1. Добавление кода TypeScript в проект примера	420
15.2. Работа с JavaScript	422
15.2.1. Включение JavaScript в процесс компиляции.	423
15.2.2. Проверка типа JavaScript-кода	423
15.3. Описание типов, используемых в коде JavaScript.	425
15.3.1. Использование комментариев для описания типов	426
15.3.2. Файлы деклараций типов	428
15.3.3. Описание стороннего JavaScript-кода.	430
15.3.4. Использование файлов деклараций проекта Definitely Typed	433
15.3.5. Использование пакетов, включающих декларации типов	435
15.4. Генерация файлов деклараций	437
Резюме	440

Часть III. Создание веб-приложений на основе TypeScript

Глава 16. Создание автономного веб-приложения, часть 1	442
16.1. Первоначальные приготовления	443
16.2. Создание инструментария	444
16.3. Добавление сборщика	445
16.4. Добавление веб-сервера разработки	447
16.5. Создание модели данных	450
16.5.1. Создание источника данных.	452
16.6. Отображение HTML-контента с использованием DOM API	455
16.6.1. Добавление поддержки стилей CSS Bootstrap	455
16.7. Использование JSX для создания HTML-контента	458
16.7.1. Как устроен рабочий процесс с JSX	460
16.7.2. Настройка компилятора и загрузчика	461
16.7.3. Создание фабричной функции	462

16.7.4. Использование класса JSX	463
16.7.5. Импорт фабричной функции в класс JSX	464
16.8. Добавление функций в приложение	465
16.8.1. Отображение отфильтрованного списка товаров	465
16.8.2. Отображение содержимого и обработка обновлений	469
Резюме	471
Глава 17. Создание автономного веб-приложения, часть 2	472
17.1. Первоначальные приготовления	473
17.2. Добавление веб-сервиса	475
17.2.1. Включение источника данных в приложение	477
17.3. Завершение работы над приложением	478
17.3.1. Добавление класса заголовка	478
17.3.2. Добавление класса деталей заказа	479
17.3.3. Добавление класса подтверждения	480
17.3.4. Финальный этап	481
17.4. Развертывание приложения	484
17.4.1. Добавление пакета рабочего варианта HTTP-сервера	484
17.4.2. Создание файла для постоянного хранения данных	485
17.4.3. Создание сервера	485
17.4.4. Использование относительных URL-адресов для запросов данных	486
17.4.5. Сборка приложения	487
17.4.6. Тестирование производственной сборки	488
17.5. Контейнеризация приложения	489
17.5.1. Установка Docker	489
17.5.2. Подготовка приложения	489
17.5.3. Создание контейнера Docker	490
17.5.4. Запуск приложения	490
Резюме	491
Глава 18. Создание приложения на Angular, часть 1	492
18.1. Первоначальные приготовления	494
18.1.1. Настройка веб-сервиса	494
18.1.2. Настройка пакета Bootstrap CSS	496
18.1.3. Запуск примера приложения	496

18.2. Роль TypeScript в разработке Angular	497
18.2.1. Конфигурация компилятора TypeScript	498
18.3. Создание модели данных	499
18.3.1. Создание источника данных.	501
18.3.2. Создание класса реализации источника данных	503
18.3.3. Настройка источника данных.	504
18.4. Отображение отфильтрованного списка товаров	505
18.4.1. Отображение кнопок категорий	507
18.4.2. Создание отображения заголовка	509
18.4.3. Объединение компонентов	509
18.5. Настройка приложения	511
Резюме	513
Глава 19. Создание приложения на Angular, часть 2	514
19.1. Первоначальные приготовления	515
19.2. Завершение работы над приложением.	516
19.2.1. Добавление компонента сводки	519
19.2.2. Создание конфигурации маршрутизации	519
19.3. Развертывание приложения	522
19.3.1. Добавление пакета производственного HTTP-сервера	522
19.3.2. Создание файла постоянного хранения данных	522
19.3.3. Создание сервера.	523
19.3.4. Использование относительных URL для запросов данных	523
19.3.5. Сборка приложения	524
19.3.6. Тестирование производственной сборки.	525
19.4. Контейнеризация приложения	526
19.4.1. Подготовка приложения	526
19.4.2. Создание контейнера Docker	526
19.4.3. Запуск приложения	527
Резюме	528
Глава 20. Создание приложения на React, часть 1	529
20.1. Первоначальные приготовления	530
20.1.1. Настройка веб-сервиса	531
20.1.2. Установка пакета Bootstrap CSS	532
20.1.3. Запуск примера приложения	533

20.2. Роль TypeScript в разработке на React.	533
20.3. Определение типов сущностей	536
20.4. Отображение отфильтрованного списка товаров	538
20.4.1. Использование функциональных компонентов и хуков	540
20.4.2. Отображение списка категорий и заголовка.	542
20.4.3. Создание и тестирование компонентов.	543
20.5. Создание хранилища данных	545
20.5.1. Реализация клиентов HTTP API.	548
Резюме	552
Глава 21. Создание приложения на React, часть 2	553
21.1. Первоначальные приготовления	554
21.2. Настройка маршрутизации URL-адресов.	555
21.3. Завершение функций примера приложения	558
21.3.1. Добавление компонента подтверждения	560
21.3.2. Использование веб-сервиса для выполнения заказов.	560
21.3.3. Завершение работы над приложением	562
21.4. Развертывание приложения	565
21.4.1. Добавление пакета производственного HTTP-сервера	565
21.4.2. Создание файла постоянных данных	566
21.4.3. Создание сервера.	566
21.4.4. Использование относительных URL-адресов для запросов данных	567
21.4.5. Сборка приложения	568
21.4.6. Тестирование производственной сборки	569
21.5. Контейнеризация приложения.	569
21.5.1. Подготовка приложения	569
21.5.2. Создание контейнера Docker	570
21.5.3. Запуск приложения	571
Резюме	572

*Посвящается моей любимой жене,
Жаки Гриффит. (А также Орешку.)*

Предисловие

Это 50-я книга, которую я написал, и третье издание «Основы TypeScript». Когда я писал первое издание, TypeScript был новинкой и мой издатель не спешил выпускать эту книгу. Я рад, что настоял на своем, поскольку глубокое погружение в технологию на начальном этапе ее становления позволяет пройти с ней бок о бок весь путь ее развития. За прошедшие годы Microsoft превратила TypeScript в мощный и надежный язык, который получил широкое признание и значительно облегчил работу с JavaScript множеству программистов. Изначально связанный с Angular, TypeScript сегодня поддерживается всеми основными средствами разработки, а его подход к улучшению JavaScript стал золотым стандартом качества.

Однако TypeScript – это не обычный язык программирования, а набор усовершенствований, применяемых к JavaScript. JavaScript – элегантный и выразительный язык, но он отличается от большинства других своим нестандартным подходом к типам данных, что вызывает бесконечную путаницу. TypeScript не меняет систему типов JavaScript, но помогает избежать неожиданных результатов. Эффективное использование TypeScript требует хорошего знания JavaScript. Воспринимайте эту книгу как учебник, благодаря которому вы сможете разобраться с наиболее запутанными особенностями JavaScript, чтобы затем начать плодотворно работать с TypeScript.

Надеюсь, что TypeScript понравится вам так же, как и мне, и эта книга даст все необходимое для создания надежных и предсказуемых JavaScript-приложений с использованием TypeScript. И конечно же, я рассчитываю на то, что мы снова встретимся в предисловии к будущему изданию «Основ TypeScript».

О книге

Книга «Основы TypeScript» призвана помочь вам освоить разработку приложений на языке TypeScript 5. Повествование начинается с настройки среды разработки и создания простого TypeScript-приложения, затем дается вводная информация о важных функциях JavaScript, после чего читатель погружается в детали того, как TypeScript преобразует JavaScript. В заключительной части книги демонстрируются три веб-приложения, написанные с помощью TypeScript: автономное приложение, приложение Angular и приложение React.

КОМУ СТОИТ ПРОЧИТАТЬ ЭТУ КНИГУ

Эта книга предназначена для опытных программистов, которые только начинают знакомиться с TypeScript, или для тех, кто приступил к разработке веб-приложений, но столкнулся с запутанностью и непредсказуемостью JavaScript.

СТРУКТУРА ИЗДАНИЯ

Книга состоит из трех частей. В первой рассказывается о том, как настроить среду разработки и создать простое веб-приложение, а также научиться пользоваться инструментами разработки.

Вторая часть посвящена описанию функций, с которыми вы будете сталкиваться каждый день, работая с TypeScript, включая базовые аннотации типов, типизированные функции, массивы, объекты и классы. В этой части также рассказывается о поддержке обобщенных типов в TypeScript, которые позволяют писать типобезопасный код без необходимости точно знать, какие типы будут использоваться во время выполнения программы, и декораторов — новой функции в TypeScript 5.

В заключительной части TypeScript демонстрируется в контексте создания одного веб-приложения тремя различными способами: полностью автономно и с помощью двух фреймворков – Angular и React. Здесь показано, как совместно используются функции, описанные во второй части.

О КОДЕ

В книге содержится большое количество примеров исходного кода. Он оформлен **вот таким шрифтом**, чтобы отделить его от обычного текста. Также вы можете заметить, что некоторый код выделен **жирным шрифтом**. Это сделано для того, чтобы обратить ваше внимание на изменения по сравнению с предыдущими листингами.

Исходный код каждой главы вы найдете по адресу <https://github.com/manningbooks/essential-typescript-5>.

Об авторе



Адам Фримен — опытный специалист в области ИТ, начавший свою карьеру с должности программиста. Занимал руководящие посты в различных компаниях, в последнее время работал техническим и исполнительным директором в одном из международных банков. Написал 50 книг по программированию, в основном посвященных разработке веб-приложений. Сейчас Адам на пенсии и проводит время за писательством и изготовлением мебели.

О научном редакторе

Фабио Клаудио Феррачати – старший консультант и ведущий аналитик/разработчик, использующий технологии Microsoft. Работает в компании TIM (www.telecomitalia.it). Имеет сертификаты Microsoft Certified Solution Developer for .NET, Microsoft Certified Application Developer for .NET, Microsoft Certified Professional, а также является известным публицистом и научным редактором. За последние десять лет он написал ряд статей для итальянских и международных журналов и стал соавтором более десяти книг по различным компьютерным темам.

Иллюстрация на обложке

Рисунок на обложке, озаглавленный Arabe, или «Араб», взят из книги Луи Курмера, опубликованной в 1841 году. Каждая иллюстрация в ней тщательно прорисована и раскрашена вручную.

В те времена по одежде можно было легко определить, где живет человек, какое у него занятие или положение в обществе. Издательство Manning отдает должное изобретательности и предпримчивости компьютерного бизнеса, используя для обложек тематических книг подобные иллюстрации. В них отражается разнообразие региональных культурных особенностей многовековой давности, которые возвращаются к людям благодаря таким изображениям, как это.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Общая информация

В этой главе

- ✓ Какие возможности предоставляет TypeScript разработчикам.
- ✓ Когда стоит использовать TypeScript в проекте.
- ✓ Ограничения TypeScript.
- ✓ Как устроена эта книга.
- ✓ Сообщения об ошибках в книге.
- ✓ Связь с автором.

TypeScript – это надстройка над JavaScript, ориентированная на создание безопасного и предсказуемого кода, который способен запускаться в любой среде выполнения JavaScript. Главная особенность TypeScript – статическая типизация – делает работу с JavaScript более предсказуемой для программистов, знакомых с такими языками, как C# и Java. В книге я объясняю, что делает TypeScript и какие возможности он предоставляет.

1.1. СТОИТ ЛИ ИСПОЛЬЗОВАТЬ TYPESCRIPT

TypeScript – это не решение всех проблем, и важно понимать, когда уместно использовать TypeScript, а когда он просто будет мешать. В последующих разделах я описываю главные возможности TypeScript и ситуации, в которых они могут быть полезны.

1.1.1. Чем полезен TypeScript для повышения производительности разработчиков

Основные возможности TypeScript направлены на повышение продуктивности разработчиков, в частности, за счет использования статических типов, которые облегчают работу с системой типов JavaScript. Другие возможности повышения производительности, такие как ключевые слова управления доступом и лаконичный синтаксис конструкторов классов, помогают предотвратить распространенные ошибки при кодировании.

Средства повышения производительности TypeScript применяются к коду JavaScript. Пакет TypeScript включает в себя компилятор, обрабатывающий файлы TypeScript и генерирующий чистый JavaScript, который можно запускать в среде выполнения JavaScript, например в Node.js или в браузере (рис. 1.1).



Рис. 1.1. Преобразование TypeScript в JavaScript-код

Благодаря сочетанию возможностей JavaScript и TypeScript язык JavaScript сохраняет гибкость и динамичность, ограничивая при этом использование типов данных, что делает их более привычными и предсказуемыми для большинства разработчиков. Это также означает, что в проектах на TypeScript все еще могут применяться сторонние пакеты JavaScript, включая поддержку использования TypeScript в готовых фреймворках для разработки приложений, описанных в части III.

Возможности TypeScript можно применять выборочно, то есть допускается использовать только те, которые подходят для конкретного проекта. Если вы новичок, то, скорее всего, захотите использовать все сразу. Однако с опытом вы научитесь работать с TypeScript более осознанно, применяя его возможности только к тем частям кода, которые особенно сложны или которые, по вашему мнению, могут вызвать проблемы.

Некоторые возможности TypeScript полностью реализуются компилятором и не оставляют следов в исполняемом JavaScript-коде. Другие возможности реализованы за счет стандартного JavaScript и выполнения дополнительных проверок при компиляции. Это означает, что для достижения лучших результатов часто необходимо вникать в работу той или иной TypeScript-функции, из-за чего они могут казаться сложными и нелогичными.

В более широком смысле TypeScript дополняет JavaScript, но не заменяет его, а разработка проекта на TypeScript в основном представляет собой процесс написания кода на JavaScript. Некоторые программисты считают, что, перейдя на

TypeScript, они смогут писать веб-приложения, не вникая, как работает JavaScript. Они видят, что TypeScript выпускает компания Microsoft, и полагают, что это C# или Java, но только для веб-разработки, однако подобное предположение приводит к путанице и разочарованию.

Чтобы эффективно работать с TypeScript, необходимо не только хорошо владеть JavaScript, но и понимать причины того или иного его поведения. В главах 3 и 4 описаны особенности JavaScript, которые необходимо знать, чтобы извлечь максимальную пользу из TypeScript и обеспечить себе прочный фундамент для понимания того, почему TypeScript является таким мощным инструментом.

Если вы готовы разбираться в системе типов JavaScript, то работа с TypeScript вам понравится. Но если вы не готовы потратить время на освоение JavaScript, то и не следует использовать TypeScript. Добавление TypeScript в проект, когда у вас нет знаний JavaScript, усложняет разработку, поскольку в таком случае придется работать с двумя наборами языковых функций, ни одна из которых не будет вести себя так, как вы ожидаете.

1.1.2. Особенности версий JavaScript

JavaScript развивался довольно хаотично, однако в последнее время наблюдается целенаправленная работа по стандартизации и совершенствованию этого языка, что привело к появлению новых возможностей, упрощающих его использование. Но и здесь не обошлось без проблем: до сих пор многие среды выполнения JavaScript не поддерживают эти современные возможности, особенно это касается старых браузеров. Из-за этого разработчикам на JavaScript приходится ограничиваться небольшим набором доступных функций языка, которые поддерживаются повсеместно. Сам по себе JavaScript непрост в освоении, а невозможность использовать функции, облегчающие разработку, делает его еще сложнее.

Компилятор TypeScript способен преобразовывать JavaScript-код, написанный с применением новейших функций, в код, соответствующий более старым версиям языка JavaScript. Это позволяет программистам использовать современные возможности JavaScript при работе с TypeScript, в то время как предыдущие версии JavaScript могут выполнять код, написанный в рамках конкретного проекта.

Компилятор TypeScript хорошо справляется с большинством языковых особенностей, однако некоторые из них не всегда возможно эффективно транслировать под устаревшие версии. Если вы работаете с самыми ранними вариациями JavaScript, то обнаружите, что каждую его современную возможность получится использовать в процессе разработки, поскольку у компилятора TypeScript попросту нет инструментов для их поддержки в устаревшем JavaScript.

При этом необходимость генерировать устаревший JavaScript-код важна не во всех проектах, поскольку компилятор TypeScript — лишь одна из частей общей цепочки инструментов. Он отвечает за внедрение возможностей TypeScript,

но результатом является современный JavaScript-код, который затем обрабатывается другими инструментами. Такой подход широко используется при разработке веб-приложений, и его примеры вы увидите в части III.

1.2. ЧТО НУЖНО ЗНАТЬ

Если вы пришли к выводу, что TypeScript подходит для вашего проекта, то важно иметь как минимум базовые знания в области разработки на JavaScript. В главах 3 и 4 я даю начальные сведения о возможностях JavaScript, которые будут полезны для понимания TypeScript, однако учтите, что данное издание не является полноценным учебником по JavaScript. В части III я демонстрирую, как можно использовать TypeScript с популярными фреймворками для разработки веб-приложений, но для этих примеров требуется знание HTML и CSS.

1.3. КАК НАСТРОИТЬ СРЕДУ РАЗРАБОТКИ

О том, как настроить среду разработки, написано в главе 2, где вы создадите свое первое приложение на TypeScript. В последующих главах могут потребоваться дополнительные пакеты. Все необходимые инструкции по их установке и использованию также будут даны.

1.4. КАКОВА СТРУКТУРА ЭТОЙ КНИГИ

Книга состоит из трех частей, каждая из которых охватывает ряд смежных тем.

Часть I «Начало работы с TypeScript». Здесь содержится информация, необходимая для начала разработки на TypeScript. Она включает в себя краткое описание того, как создавать приложения на TypeScript, а также вводную главу о важных функциях JavaScript. В главах 5 и 6 представлены средства разработки на языке TypeScript.

Часть II «Возможности TypeScript». Во второй части рассматриваются возможности языка TypeScript, которые помогают повысить продуктивность разработчиков, в том числе статические типы. TypeScript предоставляет множество различных возможностей для работы с типами, которые я подробно описываю и демонстрирую на примерах.

Часть III «Создание веб-приложений на основе TypeScript». TypeScript не используется сам по себе, поэтому часть III посвящена тому, как использовать TypeScript для создания веб-приложений с помощью наиболее популярных фреймворков. Здесь рассказывается о возможностях TypeScript, полезных для каждого фреймворка, и демонстрируются способы решения задач, обычно необходимых при разработке веб-приложений. Для того чтобы продемонстрировать способности фреймворков, я также покажу, как создать автономное веб-приложение, не зависящее от фреймворков.

1.5. МНОГО ЛИ ПРИМЕРОВ?

Примеров очень много. Лучший способ изучения TypeScript — через примеры, и в эту книгу я включил их как можно больше. Чтобы максимально увеличить количество примеров, я принял простое правило, позволяющее избегать повторного перечисления одного и того же кода или содержимого. Когда я создаю файл, я показываю его полное содержимое, как это сделано в листинге 1.1. В заголовке листинга я указываю имя файла и его папки, а внесенные мной изменения выделяю жирным шрифтом.

Листинг 1.1. Выражение неизвестного значения в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log('Number Value: ${taxValue.toFixed(2)}');
        break;
    case "string":
        console.log('String Value: ${taxValue.charAt(0)}');
        break;
    default:
        let value: never = taxValue;
        console.log('Unexpected type for value: ${value}');
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult as number;
console.log('Number value: ${myNumber.toFixed(2)})');
```

Это листинг из главы 7, в котором показано содержимое файла `index.ts`, расположенного в папке `src`. Не обращайте внимания на содержание листинга и назначение файла, просто имейте в виду, что в этом типе листинга приводится полное содержимое файла, а изменения, которые вам необходимо внести самостоятельно, чтобы следовать примеру, выделены жирным шрифтом. Некоторые файлы с кодом могут быть довольно объемными, а описываемая мною функциональность требует лишь небольших изменений. Вместо включения всего файла я использую многоточие, показывая только нужную для объяснения часть файла (листинг 1.2).

Листинг 1.2. Настройка инструментов в файле package.json из папки reactapp

```
...
"scripts": {
    "json": "json-server data.js -p 4600",
    "serve": "react-scripts start",
    "start": "npm-run-all -p serve json",
    "build": "react-scripts build",
```

```

    "test": "react-scripts test",
    "eject": "react-scripts eject"
},
...

```

Это листинг из части III, который показывает набор изменений, примененных к одной части большого файла. Когда вы увидите такой неполный листинг, знайте, что изменились только строки, выделенные жирным шрифтом, а остальная часть файла не меняется.

Иногда от вас потребуется внести изменения в разных частях файла, а это затрудняет его представление в виде частичного листинга. В такой ситуации я опускаю часть содержимого файла, как показано в листинге 1.3.

Листинг 1.3. Применение декоратора в файле abstractDataSource.ts из папки src

```

import { Product, Order } from "./entities";
import { minValue } from "../decorators";

export type ProductProp = keyof Product;

export abstract class AbstractDataSource {
    private _products: Product[];
    private _categories: Set<string>;
    public order: Order;
    public loading: Promise<void>;

    constructor() {
        this._products = [];
        this._categories = new Set<string>();
        this.order = new Order();
        this.loading = this.getData();
    }

    @minValue("price", 30)
    async getProducts(sortProp: ProductProp = "id",
        category?: string): Promise<Product[]> {
        await this.loading;
        return this.selectProducts(this._products, sortProp, category);
    }

    // ...другие методы для краткости опущены...
}

```

В этом листинге изменения по-прежнему выделены жирным шрифтом, а те части файла, которые в листинге опущены, в примере не затрагиваются.

1.6. ГДЕ МОЖНО ПОЛУЧИТЬ КОД ПРИМЕРА

Примеры проектов для всех глав книги вы можете бесплатно скачать с сайта <https://github.com/manningbooks/essential-typescript-5>. Здесь содержится все необходимое для выполнения примеров, и вам не придется набирать код вручную.

1.7. ЧТО ДЕЛАТЬ, ЕСЛИ У ВАС ВОЗНИКЛИ ПРОБЛЕМЫ С ВЫПОЛНЕНИЕМ ПРИМЕРОВ

Во-первых, следует вернуться к началу главы и сделать все заново. Большинство проблем возникает из-за пропуска какого-либо шага или неполного применения изменений, показанных в листинге. Обратите внимание на выделения в листингах, которые указывают на необходимые изменения.

Затем следует проверить список ошибок и исправлений, расположенный в репозитории книги на GitHub. Технические книги сложны, и, несмотря на все усилия автора и редакторов, ошибки неизбежны. Проверьте список исправлений, чтобы получить перечень известных ошибок и инструкции по их устранению.

Если у вас все равно возникают проблемы, то скачайте проект для главы, которую читаете, из репозитория (<https://github.com/manningbooks/essential-typescript-5>) и сравните его со своим проектом. Я писал код для репозитория GitHub, прорабатывая каждую главу, поэтому в вашем проекте должны быть те же файлы с тем же содержимым.

Если вам все же не удается заставить примеры работать, то вы можете обратиться ко мне за помощью, написав на почту adam@adam-freeman.com. Пожалуйста, укажите в письме, какую книгу вы читаете и в какой главе или примере возникла проблема. Всегда полезно указать номер страницы или листинга. Учитывайте также, что я получаю много писем и могу ответить не сразу.

1.7.1. Что делать, если вы обнаружили ошибку в книге

Вы можете сообщать мне об ошибках по электронной почте adam@adam-freeman.com, но перед тем, как это сделать, пожалуйста, проверьте список ошибок и исправлений к данной книге, который найдете в репозитории GitHub по адресу <https://github.com/manningbooks/essential-typescript-5>, — возможно, об этой ошибке уже известно.

Ошибки, которые могут привести читателей в замешательство, особенно проблемы с кодом примеров, я добавляю в файл с исправлениями в репозитории GitHub с благодарностью первому читателю, сообщившему о них. Я также публикую список менее серьезных проблем, под которыми обычно подразумеваются опечатки в тексте, сопровождающем примеры, и которые вряд ли вызовут путаницу.

1.8. КАК СВЯЗАТЬСЯ С АВТОРОМ

Вы можете связаться со мной по адресу adam@adam-freeman.com. Уже несколько лет я указываю этот адрес электронной почты в своих книгах. Изначально я сомневался в правильности такого решения, но в итоге рад, что сделал это. Я получаю письма от читателей со всего мира. Они работают или учатся в различных областях, и большинство писем очень позитивные, вежливые и приятные.

Я пытаюсь отвечать на письма оперативно, но их приходит так много, что я просто не успеваю, особенно когда погружен в работу над книгой. Я стараюсь

помочь читателям, застрявшим на примерах из книги, но прошу вас: прежде чем обратиться ко мне, выполните действия, описанные выше.

Мне приятно получать письма от читателей, но есть ряд вопросов, на которые я сразу отвечаю «нет». Боюсь, что не смогу написать код для вашего стартапа, помочь вам с заданием в университете, принять участие в споре о дизайне вашей команды разработчиков или научить вас программированию.

1.9. ЕСЛИ ВАМ ПОНРАВИЛАСЬ ЭТА КНИГА

Пожалуйста, напишите мне по адресу adam@adam-freeman.com и сообщите об этом. Мне всегда приятно читать отзывы довольных читателей, и я ценю время, которое уходит на отправку этих писем. Писать книги — нелегкий труд, и ваши сообщения стимулируют и поддерживают меня.

1.10. ЕСЛИ ВАМ НЕ ПОНРАВИЛАСЬ ЭТА КНИГА

Вы по-прежнему можете написать мне по адресу adam@adam-freeman.com, и я постараюсь вам помочь. Имейте в виду, что я смогу помочь только в том случае, если вы опишете проблему и скажете, что именно вы хотели бы, чтобы я с ней сделал. Поймите, иногда единственный выход — признать, что я просто не ваш автор. Я обязательно учту все замечания, но после 25 лет работы я пришел к выводу, что не всем нравится мое творчество.

РЕЗЮМЕ

В этой главе я объяснил, в каких случаях следует использовать TypeScript для своих проектов. Я также описал содержание и структуру книги, указал, где взять исходный код, и рассказал о том, как связаться со мной, если у вас возникнут проблемы с примерами.

- TypeScript — надстройка над JavaScript, которая для эффективного использования предполагает знание последнего.
- TypeScript не является подмножеством C#, несмотря на схожий стиль кода.
- Основной особенностью TypeScript является добавление статических типов в JavaScript.
- Компилятор TypeScript может адаптироваться к определенным версиям JavaScript, что позволяет использовать современные возможности языка в приложениях, работающих на старых платформах.

В следующей главе я расскажу о системе типов JavaScript, лежащей в основе возможностей TypeScript.

Часть I

Начало работы

с TypeScript

Ваше первое приложение на TypeScript

В этой главе

- ✓ Подготовка инструментов, необходимых для разработки на TypeScript.
- ✓ Создание и настройка проекта TypeScript.
- ✓ Компилятор TypeScript и генерация чистого JavaScript-кода.
- ✓ Запуск чистого JavaScript-кода с помощью среды выполнения Node.js.
- ✓ Подготовка проекта TypeScript для использования модулей ECMAScript.
- ✓ Установка и использование стороннего пакета JavaScript.
- ✓ Декларации типов для стороннего пакета JavaScript.

Лучший способ начать работу с TypeScript – это погрузиться в него. В данной главе я познакомлю вас с несложным процессом разработки приложения, которое отслеживает список дел. В последующих главах мы подробно рассмотрим работу функций TypeScript, но этого простого примера будет достаточно, чтобы продемонстрировать работу основных функций TypeScript. Не переживайте, если почувствуете, что не получается понять все сразу: идея состоит в том, чтобы получить общее представление о работе TypeScript и понять, какую роль он играет в приложении.

2.1. ПЕРВЫЕ ШАГИ

Для начала вам потребуется установить четыре пакета. О том, как это сделать, написано в следующих разделах. А чтобы убедиться, что пакеты работают корректно, запустите тест, предусмотренный для каждого из них.

2.1.1. Установка Node.js

Сперва загрузите и установите Node.js, также известный как Node, по адресу <https://nodejs.org/dist/v18.14.0>. Этот URL-адрес содержит инсталляторы для всех поддерживаемых платформ для версии 18.14.0, именно ее я использую в данной книге. Убедитесь, что при установке выбран менеджер пакетов Node (NPM). После завершения установки откройте окно командной строки и выполните команды, показанные в листинге 2.1, чтобы проверить работоспособность Node и NPM.

Листинг 2.1. Проверка Node и NPM

```
node --version  
npm --version
```

В результате выполнения первой команды должно получиться **v18.14.0** — это свидетельствует о том, что Node функционирует и была установлена правильная версия. Результат второй команды — **8.1.4**, что указывает на работоспособность NPM, но конкретная версия неважна.

2.1.2. Установка Git

Вторая задача — загрузить и установить инструмент управления версиями Git с сайта <https://git-scm.com/downloads>. Git не требуется непосредственно для разработки на TypeScript, но некоторые наиболее часто используемые пакеты зависят от него. После завершения установки выполните в командной строке команду, показанную в листинге 2.2. Возможно, придется вручную настроить пути к исполняемым файлам.

Листинг 2.2. Проверка Git

```
git --version
```

На момент написания книги последней версией Git для Windows и Linux является 2.39.1.

2.1.3. Установка TypeScript

Третий шаг — установка пакета TypeScript. Из командной строки выполните команду, приведенную в листинге 2.3.

Листинг 2.3. Установка пакета TypeScript

```
npm install --global typescript@5.0.2
```

После чего выполните команду, показанную в листинге 2.4, чтобы убедиться, что компилятор был установлен корректно.

Листинг 2.4. Тестирование компилятора TypeScript

```
tsc --version
```

Компилятор TypeScript называется `tsc`, и результат команды из листинга 2.4 должен показать *версию 5.0.2*.

2.1.4. Установка текстового редактора кода

Последний шаг — установка редактора кода, поддерживающего TypeScript. На сегодняшний день к таким относится большинство популярных редакторов, однако если у вас нет особых предпочтений, то я рекомендую воспользоваться Visual Studio Code (<https://code.visualstudio.com>). Visual Studio Code — это бесплатный кроссплатформенный редактор кода с открытым исходным кодом, и все примеры для этой книги я написал с его помощью.

Для запуска Visual Studio Code выполните команду `code` или щелкните на значке, созданном при установке программы, и вы увидите такое окно приветствия, как на рис. 2.1. (Возможно, перед выполнением команды `code` из командной строки вам потребуется указать путь к исполняемому файлу Visual Studio Code.)

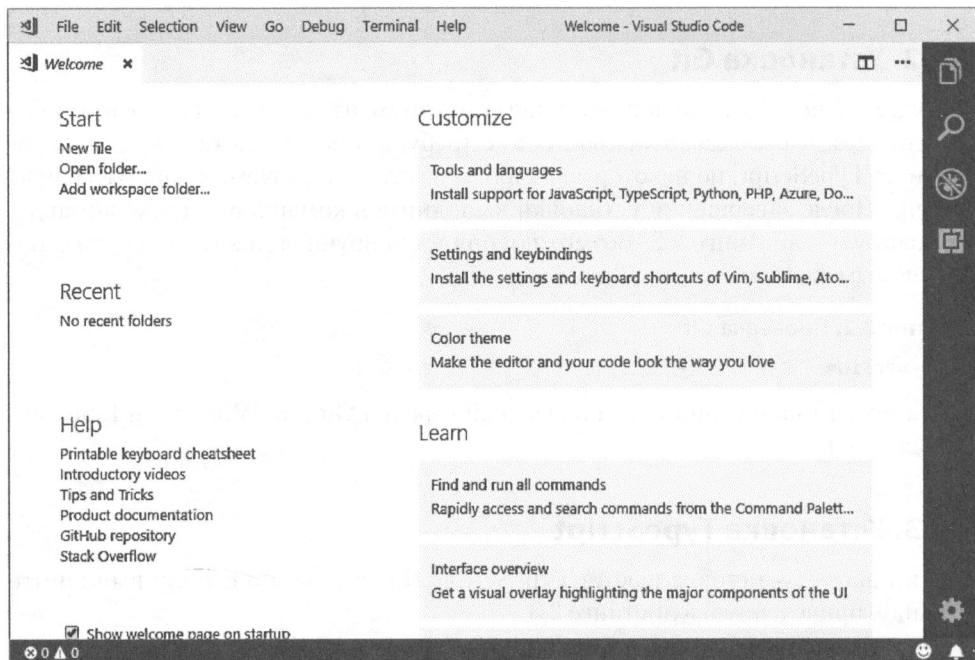


Рис. 2.1. Экран приветствия Visual Studio Code

СОВЕТ

В некоторых редакторах можно указывать версию TypeScript, отличную от той, которая содержится в проекте. Обычно это приводит к появлению ошибок в редакторе кода, даже если средства командной строки показывают успешную компиляцию. Например, если вы используете Visual Studio Code, то при редактировании файла TypeScript в правой нижней части окна редактора будет отображаться используемая версия TypeScript. Щелкните на этой надписи, нажмите кнопку Select TypeScript Version и выберите нужную версию.

2.2. СОЗДАНИЕ ПРОЕКТА

Теперь, когда все необходимые приготовления завершены, а средства разработки установлены, предлагаю приступить к практике и создать простое приложение для составления списка дел. TypeScript чаще всего используется для разработки веб-приложений, что я продемонстрирую на примере популярных фреймворков в части III. Однако в текущей главе мы напишем консольное приложение, чтобы фокусироваться на TypeScript и избежать сложностей, связанных с фреймворком для веб-приложений.

Наше приложение будет отображать список имеющихся задач, позволит создавать новые задачи, отмечать существующие как выполненные, а также будет доступен фильтр для отображения списка уже выполненных задач. Когда базовые функции будут реализованы, мы добавим возможность постоянного хранения данных для предотвращения потери изменений при завершении работы приложения.

2.2.1. Инициализация проекта

Чтобы подготовить папку проекта для этой главы, откройте командную строку, выберите подходящее место и создайте там папку с именем `todo`. Выполните команды, приведенные в листинге 2.5, чтобы перейти в эту папку и инициализировать ее для разработки.

Листинг 2.5. Инициализация папки проекта

```
cd todo  
npm init --yes
```

Команда `npm init` создает файл `package.json`, который используется для отслеживания пакетов, необходимых проекту, а также для настройки средств разработки.

2.2.2. Создание файла конфигурации компилятора

Пакет TypeScript, установленный кодом из листинга 2.3, включает в себя компилятор с именем `tsc`, который компилирует код TypeScript в чистый JavaScript. Чтобы определить конфигурацию компилятора, создайте в папке `todo` файл `tsconfig.json` с содержимым листинга 2.6.

Листинг 2.6. Содержимое файла tsconfig.json из папки todo

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "CommonJS"
  }
}
```

Компилятор TypeScript подробно описан в главе 5, а настройки выше задают следующие параметры: использовать последнюю версию JavaScript, файлы TypeScript проекта располагаются в папке `src`, результат должен быть помещен в папку `dist`, при загрузке кода из отдельных файлов следует применять настройку `CommonJS`.

2.2.3. Добавление файла с кодом TypeScript

Файлы с кодом TypeScript имеют расширение `ts`. Чтобы добавить его в проект, создайте папку `todo/src` и добавьте в нее файл `index.ts` с кодом, показанным в листинге 2.7. Это соответствует общепринятой традиции называть главный файл приложения `index`, а расширение `ts` указывает на то, что он содержит код JavaScript.

Листинг 2.7. Содержимое файла index.ts в папке src

```
console.clear();
console.log("Adam's Todo List");
```

Файл содержит обычные операторы JavaScript, которые используют объект `console` для очистки окна командной строки и вывода простого сообщения. Этого достаточно для того, чтобы убедиться, что все работает.

2.2.4. Компиляция и выполнение кода

Файлы TypeScript необходимо скомпилировать для получения чистого JavaScript-кода, который может быть выполнен браузерами или средой исполнения Node.js, установленной в начале этой главы. Из командной строки запустите компилятор в папке `todo`, используя команду из листинга 2.8.

Листинг 2.8. Запуск компилятора TypeScript

```
tsc
```

Компилятор считывает настройки конфигурации из файла `tsconfig.json`, находит файлы TypeScript в каталоге `src` и создает папку `dist` для записи JavaScript-кода. В результате в папке `dist` появляется файл `index.js`, расширение `js` которого указывает на то, что файл содержит JavaScript-код. Если изучить содержимое файла, то можно увидеть в нем следующие выражения:

```
console.clear();
console.log("Adam's Todo List");
```

Файлы TypeScript и JavaScript содержат одни и те же утверждения, поскольку я пока не использовал никаких возможностей TypeScript. По мере формирования приложения содержимое файла TypeScript и сгенерированных JavaScript-файлов начнет различаться.

ВНИМАНИЕ

Не изменяйте файлы в папке `dist`, поскольку они будут перезаписаны при следующем запуске компилятора. В TypeScript-разработке изменения вносятся в файлы с расширением `ts`, которые затем компилируются в файлы JavaScript с расширением `js`.

Чтобы выполнить скомпилированный код, в папке `todo` с помощью командной строки выполните команду из листинга 2.9.

Листинг 2.9. Выполнение скомпилированного кода

```
node dist/index.js
```

Команда `node` запускает среду выполнения Node.js JavaScript, а в качестве аргумента указывается файл, содержимое которого должно быть выполнено. Если средства разработки были установлены корректно, консольное окно очистится и отобразит следующий результат:

```
Adam's Todo List
```

2.2.5. Определение модели данных

В приведенном примере разрабатывается приложение для управления списком дел. Пользователь сможет просматривать список, добавлять в него новые пункты, отмечать уже выполненные пункты и фильтровать их. В этом разделе я начну использовать TypeScript для определения модели данных, которая описывает данные приложения и операции, которые можно с ними выполнять. Для начала добавьте в папку `src` файл `todoItem.ts` со следующим кодом (листинг 2.10).

Листинг 2.10. Содержимое файла `todoItem.ts` из папки `src`

```
export class TodoItem {
    public id: number;
    public task: string;
    public complete: boolean = false;

    public constructor(id: number, task: string,
                      complete: boolean = false) {
        this.id = id;
        this.task = task;
        this.complete = complete;
    }

    public printDetails(): void {
        console.log(`${this.id}\t${this.task} ${this.complete
            ? "\t(complete)": ""}`);
    }
}
```

Классы – это шаблоны, описывающие тип данных. Более подробно классы описаны в главе 4, но код из листинга 2.10 будет понятен любому программисту с опытом работы на языках C# или Java, хотя некоторые детали могут быть не сразу очевидны.

Классом в листинге 2.10 является `TodoItem`. У него есть свойства `id`, `task` и `complete`, а также метод `printDetails`, выводящий на консоль краткую информацию об элементе списка дел. TypeScript построен на JavaScript, поэтому код в листинге 2.10 – это смесь базовых возможностей JavaScript и улучшений, характерных для TypeScript. Например, JavaScript поддерживает работу с классами, их конструкторами, свойствами и методами. Однако такие возможности, как ключевые слова управления доступом (например, `public`), предоставляет TypeScript. Главной особенностью TypeScript является статическая типизация. Она позволяет указать тип каждого свойства и параметра в классе `TodoItem`. Например, так:

```
...
public id: number;
...

```

Это пример *аннотации типов*, которая сообщает компилятору TypeScript, что свойству `id` можно присваивать значения только типа `number`. В главе 3 я объясняю, что в JavaScript существует гибкий подход к типизации данных и одно из преимуществ TypeScript заключается в том, что типы данных становятся более совместимыми с другими языками программирования, сохраняя возможность использовать привычный для JavaScript подход к типизации там, где это необходимо.

СОВЕТ

Не волнуйтесь, если вы еще не знакомы с тем, как JavaScript работает с типами данных. В главах 3 и 4 вы найдете всю необходимую информацию о возможностях JavaScript, которые нужно знать для эффективного использования TypeScript.

Я написал класс в листинге 2.10, чтобы подчеркнуть сходство TypeScript с такими языками, как C# и Java, но обычно классы в TypeScript определяются иначе. В листинге 2.11 класс `TodoItem` переделан с использованием лаконичного синтаксиса TypeScript.

Листинг 2.11. Использование более лаконичного кода в файле `todoItem.ts` из папки `src`

```
export class TodoItem {

    constructor(public id: number,
                public task: string,
                public complete: boolean = false) {
        // не требуется никаких операторов
    }

    printDetails(): void {
        console.log(`${this.id}\t${this.task} ${this.complete
            ? '\t(complete)": "'"}');
    }
}
```

Поддержка статических типов данных — это лишь малая часть глобальной цели TypeScript: сделать код JavaScript более безопасным и предсказуемым. Лаконичный синтаксис конструктора из листинга 2.11 позволяет классу `TodoItem` получать параметры и использовать их для создания свойств экземпляра за один шаг. Благодаря этому исключается подверженный ошибкам процесс определения свойства и явного присвоения ему значения, полученного с помощью параметра.

Изменение метода `printDetails` удаляет ключевое слово управления доступом `public`. Оно больше не нужно, поскольку в TypeScript предполагается, что все методы и свойства являются общедоступными, если не используется другой уровень доступа. (Ключевое слово `public` по-прежнему используется в конструкторе. Именно так компилятор TypeScript распознает, что используется лаконичный синтаксис конструктора, о чем сказано в главе 11.)

Создание класса коллекции элементов todo

Следующим шагом будет создание класса, который будет собирать вместе пункты списка дел, чтобы ими было удобнее управлять. Добавьте в папку `src` файл с именем `todoCollection.ts` с кодом, показанным в листинге 2.12.

Листинг 2.12. Содержимое файла `todoCollection.ts` из папки `src`

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
    private nextId: number = 1;

    constructor(public userName: string,
                public todoItems: TodoItem[] = []) {
        // не требуется никаких операторов
    }

    addTodo(task: string): number {
        while (this.getTodoById(this.nextId)) {
            this.nextId++;
        }
        this.todoItems.push(new TodoItem(this.nextId, task));
        return this.nextId;
    }

    getTodoById(id: number) : TodoItem {
        return this.todoItems.find(item => item.id === id);
    }

    markComplete(id: number, complete: boolean) {
        const todoItem = this.getTodoById(id);
        if (todoItem) {
            todoItem.complete = complete;
        }
    }
}
```

Проверка основных функций модели данных

Прежде чем двигаться дальше, необходимо убедиться, что базовые функции класса `TodoCollection` работают так, как задумано. В главе 6 говорится о том, как проводить модульное тестирование проектов на TypeScript, но для этой главы будет достаточно создать несколько объектов `TodoItem` и сохранить их в объекте `TodoCollection`. В листинге 2.13 заменен код в файле `index.ts`, удалены операторы заполнителя, добавленные в начале главы.

Листинг 2.13. Тестирование модели данных в файле `index.ts` из папки `src`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos = [
  new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
  new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection = new TodoCollection("Adam", todos);

console.clear();
console.log(`${collection.userName}'s Todo List');

let newId = collection.addTodo("Go for run");
let todoItem = collection.getTodoById(newId);
todoItem.printDetails();
```

Код, приведенный в листинге 2.13, использует возможности чистого JavaScript. Операторы `import` нужны для объявления зависимостей от классов `TodoItem` и `TodoCollection` и являются частью функции JavaScript-модулей, которая позволяет определять код в нескольких файлах (подробнее об этом – в главе 4). Определение массива и использование ключевого слова `new` для создания экземпляров классов также являются базовыми возможностями, равно как и обращение к объекту `console`.

ПРИМЕЧАНИЕ

В коде из листинга 2.13 используются функции, недавно появившиеся в языке JavaScript. Как будет объяснено в главе 5, компилятор TypeScript позволяет легко использовать современные возможности JavaScript, такие как ключевое слово `let`, даже если они не поддерживаются средой выполнения JavaScript, в которой планируется выполнять код, например старыми браузерами. Функции JavaScript, понимание которых необходимо для эффективной разработки на TypeScript, описаны в главах 3 и 4.

Компилятор TypeScript призван помочь разработчикам, но при этом не мешать им. В процессе компиляции он анализирует используемые типы данных и информацию о типах, которую я описал в классах `TodoItem` и `TodoCollection`. Основываясь на этих данных, компилятор может определить типы данных в листинге 2.13. В результате получается код, не содержащий явной статической информации о типах, однако он все еще может быть проверен компилятором на предмет безопасности типов. Для демонстрации подобного процесса в файл `index.ts` был добавлен оператор (листинг 2.14).

Листинг 2.14. Добавление оператора в файл index.ts

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos = [
    new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
    new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection = new TodoCollection("Adam", todos);

console.clear();
console.log(`${collection.userName}'s Todo List');

let newItem = collection.addTodo("Go for run");
let todoItem = collection.getTodoById(newItemId);
todoItem.printDetails();
collection.addTodo(todoItem);
```

Новый оператор вызывает метод `TodoCollection.addTodo`, используя в качестве аргумента объект `TodoItem`. Компилятор анализирует, как метод `addTodo` определяется в файле `todoItem.ts`, и обнаруживает, что метод ожидает получить данные другого типа.

```
...
addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
        this.nextId++;
    }
    this.todoItems.push(new TodoItem(this.nextId, task));
    return this.nextId;
}
...
```

Информация о типе метода `addTodo` говорит компилятору TypeScript, что параметр `task` должен быть типа `string`, а возвращаемое значение — `number`. (Типы `string` и `number` являются встроенными функциями JavaScript и описаны в главе 3.) Для компиляции кода в папке `todo` выполните команду, показанную в листинге 2.15.

Листинг 2.15. Запуск компилятора

```
tsc
```

Компилятор TypeScript обрабатывает код проекта, обнаруживает, что значение параметра, используемое для вызова метода `addTodo`, не соответствует типу данных, и выдает следующую ошибку:

```
src/index.ts:16:20 - error TS2345: Argument of type 'TodoItem' is not
  assignable to parameter of type 'string'.
16 collection.addTodo(todoItem);
~~~~~
Found 1 error in src/index.ts:16
```

TypeScript отлично справляется с задачей понимания происходящего и выявления проблем, позволяя вам добавлять в проект столько информации о типах,

сколько нужно. В книге я постарался предоставить максимум информации о типах для лучшего понимания листингов примеров, поскольку многие примеры здесь связаны с тем, как компилятор TypeScript обрабатывает типы данных. В листинге 2.16 к коду в файле `index.ts` добавлена информация о типах и отключен оператор, вызывающий ошибку компилятора.

Листинг 2.16. Добавление информации о типе в файл index.ts

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
    new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
    new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`${collection.userName}'s Todo List');

let newId: number = collection.addTodo("Go for run");
let todoItem: TodoItem = collection.getTodoById(newId);
todoItem.printDetails();
//collection.addTodo(todoItem);
```

Информация о типе, добавленная в листинге 2.16, не меняет принцип работы кода, но делает используемые типы данных более явными, что может облегчить понимание кода и избавить компилятор от необходимости определять типы данных самостоятельно. Чтобы скомпилировать и выполнить код, воспользуйтесь командами из листинга 2.17 в папке `todo`.

Листинг 2.17. Компиляция и выполнение

```
tsc
node dist/index.js
```

После выполнения этого кода будет получен следующий результат:

```
Adam's Todo List
5      Go for run
```

2.2.6. Добавление функций в класс коллекции

Следующим шагом будет добавление новых функций в класс `TodoCollection`. Сперва мы изменим способ хранения объектов `TodoItem`, чтобы использовать JavaScript `Map` (листинг 2.18).

Листинг 2.18. Использование map в файле todoCollection.ts из папки src

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
    private nextId: number = 1;
    private itemMap = new Map<number, TodoItem>();
```

```
constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
}

addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
        this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
}

getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
}

markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
        todoItem.complete = complete;
    }
}
}
```

TypeScript поддерживает обобщенные типы, которые представляют собой заполнители для типов, разрешаемых при создании объекта. Например, JavaScript Map – это универсальная коллекция, в которой хранятся пары «ключ/значение». Благодаря подобной динамичности типов JavaScript в Map можно хранить любой набор типов данных со всевозможным набором ключей. Чтобы установить ограничения на типы в Map в листинге 2.18, были использованы аргументы обобщенных типов, указывающие компилятору TypeScript допустимые типы ключей и значений.

```
...
private itemMap = new Map<number, TodoItem>();
...
```

Аргументы обобщенного типа заключаются в угловые скобки (символы `<` и `>`), и для Map в листинге 2.18 как раз они и задаются. Они сообщают компилятору, что в Map будут храниться объекты TodoItem, использующие в качестве ключей значения типа `number`. Компилятор выдаст ошибку, если оператор попытается сохранить в Map тип данных или использовать ключ, отличный от `number`. Обобщенные типы являются важной особенностью TypeScript и подробно рассматриваются в главе 12.

Предоставление доступа к элементам списка дел

Класс TodoCollection определяет метод `getTodoById`, но приложению необходимо отобразить список элементов с учетом заданных фильтров, чтобы исключить завершенные задачи. В листинге 2.19 показан метод, предоставляющий доступ к объектам `TodoItem`, которыми управляет `TodoCollection`.

Листинг 2.19. Предоставление доступа к элементам в файле todoCollection.ts из папки src

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
    private nextId: number = 1;
    private itemMap = new Map<number, TodoItem>();

    constructor(public userName: string, todoItems: TodoItem[] = []) {
        todoItems.forEach(item => this.itemMap.set(item.id, item));
    }

    addTodo(task: string): number {
        while (this.getTodoById(this.nextId)) {
            this.nextId++;
        }
        this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
        return this.nextId;
    }

    getTodoById(id: number): TodoItem {
        return this.itemMap.get(id);
    }

    getTodoItems(includeComplete: boolean): TodoItem[] {
        return [...this.itemMap.values()]
            .filter(item => includeComplete || !item.complete);
    }

    markComplete(id: number, complete: boolean) {
        const todoItem = this.getTodoById(id);
        if (todoItem) {
            todoItem.complete = complete;
        }
    }
}
```

Метод `getTodoItems` получает объекты из `Map` с помощью метода `values` и применяет их для создания массива с использованием оператора расширения JavaScript, который представляет собой многоточие (...). Объекты обрабатываются с помощью метода `filter` для выбора необходимых объектов, при этом параметр `includeComplete` указывает, какие именно объекты необходимы. Компилятор TypeScript использует предоставленную ему информацию, чтобы проследить, как типы проходят каждый этап.

Аргументы обобщенного типа, используемые для создания `Map`, сообщают компилятору, что она содержит объекты `TodoItem`, поэтому компилятор знает, что метод `values` вернет объекты `TodoItem` и что они также будут типами объектов в массиве. После этого компилятор знает, что функция, переданная методу `filter`, будет обрабатывать объекты `TodoItem` и каждый из этих объектов будет определять свойство `complete`. Если я попытаюсь прочитать свойство или метод, которые не определены

классом `TodoItem`, то компилятор TypeScript сообщит об ошибке. Аналогичным образом, компилятор выдал бы ошибку, если бы результат оператора `return` был несовместим с типом результата, указанным в классе `TodoItem`.

В листинге 2.20 продемонстрирован обновленный код `index.ts`, в котором используется новая функция класса `TodoCollection` для отображения простого списка дел пользователя.

Листинг 2.20. Получение элементов коллекции в файле `index.ts` из папки `src`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
  new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`${collection.userName}'s Todo List');

//let newItem: number = collection.addTodo("Go for run");
//let todoItem: TodoItem = collection.getTodoById(newId);
//todoItem.printDetails();
//collection.addTodo(todoItem);
collection.getTodoItems(true).forEach(item => item.printDetails());
```

Новый оператор вызывает метод `getTodoItems`, определенный в листинге 2.19, и использует стандартный метод JavaScript `forEach` для записи описания каждого объекта `TodoItem` с помощью объекта `console`.

Выполните в папке `todo` команды, показанные в листинге 2.21, для компиляции и запуска кода.

Листинг 2.21. Компиляция и выполнение

```
tsc
node dist/index.js
```

После выполнения этого кода будет выдан следующий результат:

```
Adam's Todo List
1      Buy Flowers
2      Get Shoes
3      Collect Tickets
4      Call Joe      (complete)
```

Удаление выполненных заданий

По мере добавления задач и отметок об их выполнении количество элементов в коллекции будет расти, что может затруднить управление ими для пользователя. В листинге 2.22 продемонстрирован метод, удаляющий завершенные элементы из коллекции.

Листинг 2.22. Удаление элементов, помеченных как выполненные, из файла todoCollection.ts в папке src

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
    private nextId: number = 1;
    private itemMap = new Map<number, TodoItem>();

    constructor(public userName: string, todoItems: TodoItem[] = []) {
        todoItems.forEach(item => this.itemMap.set(item.id, item));
    }

    addTodo(task: string): number {
        while (this.getTodoById(this.nextId)) {
            this.nextId++;
        }
        this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
        return this.nextId;
    }

    getTodoById(id: number) : TodoItem {
        return this.itemMap.get(id);
    }

    getTodoItems(includeComplete: boolean): TodoItem[] {
        return [...this.itemMap.values()]
            .filter(item => includeComplete || !item.complete);
    }

    markComplete(id: number, complete: boolean) {
        const todoItem = this.getTodoById(id);
        if (todoItem) {
            todoItem.complete = complete;
        }
    }

    removeComplete() {
        this.itemMap.forEach(item => {
            if (item.complete) {
                this.itemMap.delete(item.id);
            }
        })
    }
}
```

Метод `removeComplete` использует метод `Map.forEach` для проверки каждого `TodoItem`, хранящегося в `Map`, и вызывает метод `delete` для тех, у которых свойство `complete` равно `true`. В листинге 2.23 обновленный код в файле `index.ts` вызывает новый метод.

Листинг 2.23. Тестирование удаления элементов в файле index.ts из папки src

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
```

```

let todos: TodoItem[] = [
  new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
  new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`${collection.userName}'s Todo List');

//let newItem: number = collection.addTodo("Go for run");
//let todoItem: TodoItem = collection.getTodoById(newId);
//todoItem.printDetails();
//collection.addTodo(todoItem);
collection.removeComplete();
collection.getTodoItems(true).forEach(item => item.printDetails());

```

Для компиляции и запуска кода выполните команды из листинга 2.24 в папке todo.

Листинг 2.24. Компиляция и выполнение

```
tsc
node dist/index.js
```

Будет получен следующий результат, показывающий, что выполненная задача удалена из коллекции:

```

Adam's Todo List
1      Buy Flowers
2      Get Shoes
3      Collect Tickets

```

Подсчет количества элементов списка

Последняя функция, необходимая для класса TodoCollection, — это подсчет общего количества объектов TodoItem, а также числа завершенных и еще не завершенных дел.

В предыдущих листингах я использовал классы, поскольку большинство программистов привыкли действовать именно так, создавая типы данных. Объекты JavaScript также могут быть определены с помощью литерального синтаксиса, для них TypeScript может проверять и обеспечивать статические типы так же, как и для объектов, созданных на основе классов. При работе с литералами объектов компилятор TypeScript учитывает комбинацию имен свойств и типов их значений. Эта комбинация называется *формой объекта*, а конкретная комбинация имен и типов — *структурой типа*. Метод, возвращающий объект с информацией об элементах коллекции, добавлен в класс TodoCollection в листинге 2.25.

Листинг 2.25. Использование структуры типа в файле todoCollection.ts из папки src

```

import { TodoItem } from "./todoItem";

type ItemCounts = {
  total: number,
  incomplete: number
}

```

```

export class TodoCollection {
    private nextId: number = 1;
    private itemMap = new Map<number, TodoItem>();

    constructor(public userName: string, todoItems: TodoItem[] = []) {
        todoItems.forEach(item => this.itemMap.set(item.id, item));
    }

    addTodo(task: string): number {
        while (this.getTodoById(this.nextId)) {
            this.nextId++;
        }
        this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
        return this.nextId;
    }

    getTodoById(id: number): TodoItem {
        return this.itemMap.get(id);
    }

    getTodoItems(includeComplete: boolean): TodoItem[] {
        return [...this.itemMap.values()]
            .filter(item => includeComplete || !item.complete);
    }

    markComplete(id: number, complete: boolean) {
        const todoItem = this.getTodoById(id);
        if (todoItem) {
            todoItem.complete = complete;
        }
    }

    removeComplete() {
        this.itemMap.forEach(item => {
            if (item.complete) {
                this.itemMap.delete(item.id);
            }
        })
    }

    getItemCounts(): ItemCounts {
        return {
            total: this.itemMap.size,
            incomplete: this.getTodoItems(false).length
        };
    }
}

```

Ключевое слово `type` используется для создания *псевдонима типа*, который представляет собой удобный способ присвоения имени структуре типа. Псевдоним типа из листинга 2.25 описывает объекты с двумя свойствами типа `number`, названными `total` и `incomplete`. Псевдоним типа используется в качестве результата метода `getItemCounts`, который с помощью синтаксиса литерала объекта JavaScript создает объект с формой, соответствующей псевдониму. В листинге 2.26 файл `index.ts` обновлен таким образом, чтобы пользователь мог видеть количество незавершенных элементов.

Листинг 2.26. Отображение количества элементов в файле index.ts

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
  new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
//console.log(`${collection.userName}'s Todo List');
console.log(`${collection.userName}'s Todo List '
  + `(${collection.getItemCounts().incomplete} items to do)'`);

//collection.removeComplete();
collection.getTodoItems(true).forEach(item => item.printDetails());
```

Для компиляции и запуска кода выполните команды из листинга 2.27 в папке todo.

Листинг 2.27. Компиляция и выполнение

```
tsc
node dist/index.js
```

После этого будет получен следующий результат:

```
Adam's Todo List (3 items to do)
1      Buy Flowers
2      Get Shoes
3      Collect Tickets
4      Call Joe      (complete)
```

2.3. ИСПОЛЬЗОВАНИЕ ПАКЕТОВ СТОРОННИХ ПРОИЗВОДИТЕЛЕЙ

Базовые функции уже реализованы, но есть возможности для улучшения. Одно из преимуществ работы с JavaScript-кодом — наличие экосистемы пакетов, которые можно включать в проекты. TypeScript поддерживает любые JavaScript-пакеты, но с добавлением поддержки статических типов. Для того чтобы запрашивать у пользователя команды и обрабатывать ответы, мы будем использовать замечательный пакет Inquirer.js (<https://github.com/SBoudrias/Inquirer.js>).

2.3.1. Подготовка к работе со сторонним пакетом

Одним из недостатков написания JavaScript-кода является множество конкурирующих стандартов распространения и использования пакетов. Когда JavaScript только появился, единого формата пакетов не существовало, и возникло несколько конкурирующих стандартов. В настоящее время спецификация языка JavaScript включает общий стандарт для модулей, называемый *модулями ECMAScript*. Большинство сред выполнения JavaScript, включая Node.js, реализуют поддержку

ECMAScript, и большинство популярных пакетов JavaScript обновляются, чтобы публиковаться в этом формате.

TypeScript поддерживает модули ECMAScript, но для включения этой возможности требуется внести некоторые изменения в проект. В листинге 2.28 в файл `package.json` добавлено конфигурационное свойство, которое указывает, что данному проекту необходима поддержка модулей ECMAScript.

Листинг 2.28. Добавление свойства конфигурации в файл `package.json` из папки todo

```
{
  "name": "todo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "module"
}
```

В листинге 2.29 изменена конфигурация компилятора TypeScript таким образом, чтобы он искал свойство `type` в файле `package.json` для определения типа используемых модулей.

Листинг 2.29. Настройка компилятора в файле `tsconfig.json` из папки todo

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "Node16"
  }
}
```

До сих пор нам удавалось объявлять зависимости между файлами кода без указания расширений файла, как, например, в этом операторе из файла `todoCollection.ts`:

```
...
import { TodoItem } from "./todoItem";
...
```

Более подробно операторы `import` описаны в главе 4. Однако для данной главы важно то, что имя файла указано без расширения. Но способ, которым в Node.js реализованы модули ECMAScript, требует включения расширения файла, как показано в листинге 2.30.

Листинг 2.30. Добавление расширения файла в файл `todoCollection.ts` из папки `src`

```
import { TodoItem } from "./todoItem.js";

type ItemCounts = {
  total: number,
```

```

    incomplete: number
}
...

```

Странность здесь заключается в том, что в операторе `import` должен быть указан JavaScript-файл, который будет сгенерирован из файла TypeScript. На это есть свои причины, которые мы разберем в последующих главах. Такое же изменение требуется внести в операторы `import` в файле `index.ts`, как показано в листинге 2.31.

Листинг 2.31. Добавление расширений файлов в файл `index.ts` из папки `src`

```

import { TodoItem } from "./todoItem.js";
import { TodoCollection } from "./todoCollection.js";

let todos: TodoItem[] = [
    new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
    new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];
...

```

2.3.2. Установка и использование

сторонних пакетов

Чтобы добавить `Inquirer.js` в проект, выполните команду из листинга 2.32 в папке `todo`.

Листинг 2.32. Добавление пакета в проект

```
npm install inquirer@9.1.4
```

Пакеты добавляются в TypeScript-проекты так же, как и в проекты на чистом JavaScript, — с помощью команды `npm install`. Чтобы начать работу с новым пакетом, добавьте в файл `index.ts` операторы, показанные в листинге 2.33.

Листинг 2.33. Подключение нового пакета в файле `index.ts`

```

import { TodoItem } from "./todoItem.js";
import { TodoCollection } from "./todoCollection.js";
import inquirer from "inquirer";

let todos: TodoItem[] = [
    new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
    new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

function displayTodoList(): void {
    console.log(`${collection.userName}'s Todo List '
        + `${collection.getItemCounts().incomplete} items to do'`);
    collection.getTodoItems(true).forEach(item => item.printDetails());
}

enum Commands {
    Quit = "Quit"
}

```

```

function promptUser(): void {
    console.clear();
    displayTodoList();
    inquirer.prompt({
        type: "list",
        name: "command",
        message: "Choose option",
        choices: Object.values(Command)
    }).then(answers => {
        if (answers["command"] !== Commands.Quit) {
            promptUser();
        }
    })
}

promptUser();

```

TypeScript не препятствует использованию JavaScript-кода. В листинге 2.33 используется пакет Inquirer.js, позволяющий выдавать подсказки пользователю и предлагать выбор команд. Сейчас доступна только одна команда — `Quit`, но в ближайшее время мы добавим больше полезных функций.

СОВЕТ

В этой книге нет детального описания API Inquirer.js, поскольку он не имеет прямого отношения к TypeScript. Если вы хотите использовать Inquirer.js в собственных проектах, перейдите по ссылке <https://github.com/SBoudrias/Inquirer.js>.

Метод `inquirer.prompt` используется для запроса ответа от пользователя и настраивается с помощью JavaScript-объекта. Выбранные нами варианты конфигурации предоставляют пользователю список, по которому можно перемещаться с помощью клавиш со стрелками, а чтобы сделать выбор — нажать `Enter`. Когда пользователь выбирает команду, вызывается функция, переданная в метод `then`, и выбор становится доступным через свойство `answers.command`.

В листинге 2.33 показано, как можно органично использовать код TypeScript и JavaScript из пакета Inquirer.js. Ключевое слово `enum` — это функция TypeScript, которая позволяет присваивать имена значениям, как описано в главе 9. Это дает возможность определять команды и обращаться к ним без необходимости дублировать строковые значения в приложении. Значения из `enum` используются вместе с функциями Inquirer.js, например, так:

```

...
if (answers["command"] !== Commands.Quit) {
...

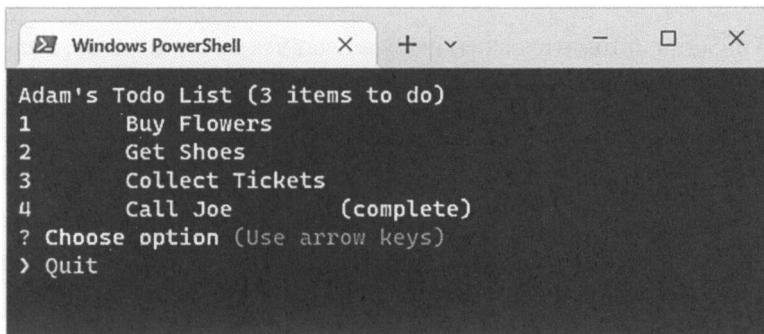
```

Для компиляции и запуска кода выполните команды из листинга 2.34 в папке `todo`.

Листинг 2.34. Компиляция и выполнение

```
tsc
node dist/index.js
```

После выполнения кода на экран будет выведен список дел, а также запрос на выбор команды, как показано на рис. 2.2, хотя доступна только одна команда — `Quit`.



```
Adam's Todo List (3 items to do)
1     Buy Flowers
2     Get Shoes
3     Collect Tickets
4     Call Joe      (complete)
? Choose option (Use arrow keys)
> Quit
```

Рис. 2.2. Выдача пользователю команды

Если нажать клавишу `Enter`, то будет выбрана команда `Quit` и приложение завершит свою работу.

2.3.3. Добавление деклараций типов для пакета JavaScript

TypeScript не препятствует использованию JavaScript-кода, но и не может помочь при его использовании. У компилятора нет сведений о типах данных, содержащихся в `Inquirer.js`, и он вынужден полагаться на то, что я применяю правильные типы аргументов для запроса пользователю и безопасно обрабатываю объекты ответа.

Существует два способа предоставить TypeScript информацию, необходимую для статической типизации. Первый способ — описать типы самостоятельно. О возможностях, которые предоставляет TypeScript для описания JavaScript-кода, вы узнаете в главе 14. Описать JavaScript-код несложно, но это требует времени и глубокого понимания описываемого кода.

Второй подход заключается в использовании деклараций (объявлений) типов, предоставленных кем-либо другим. Проект Definitely Typed представляет собой репозиторий деклараций типов TypeScript для множества пакетов JavaScript, включая `Inquirer.js`. Чтобы установить декларации типов, выполните команду из листинга 2.35 в папке `todo`.

Листинг 2.35. Установка определений типов

```
npm install --save-dev @types/inquirer@9.0.3
```

Декларации типов устанавливаются с помощью команды `npm install`, как и пакеты JavaScript. Аргумент `--save-dev` необходим для пакетов, которые используются в разработке, но не являются частью приложения. За `@types/` следует название пакета, для которого требуется описание типов. Для пакета `Inquirer.js` пакет деклараций типов имеет вид `@types/inquirer`, поскольку `inquirer` — это имя, используемое для установки пакета JavaScript.

ПРИМЕЧАНИЕ

Подробности о проекте Definitely Typed и пакетах, для которых доступны декларации типов, см. на сайте <https://github.com/DefinitelyTyped/DefinitelyTyped>.

Компилятор TypeScript автоматически обнаруживает декларации типов, а пакет, установленный командой из листинга 2.35, позволяет компилятору проверить типы данных, используемые в API Inquirer.js. Чтобы продемонстрировать эффект декларации типов, в листинге 2.36 приводится свойство конфигурации, которое не поддерживается Inquirer.js.

Листинг 2.36. Добавление новой конфигурации в файл index.ts из папки src

```
...
function promptUser(): void {
    console.clear();
    inquirer.prompt({
        type: "list",
        name: "command",
        message: "Choose option",
        choices: Object.values(Command),
        badProperty: true
    }).then(answers => {
        // не требуется никаких действий
        if (answers["command"] !== Commands.Quit) {
            promptUser();
        }
    })
}
...

```

В API Inquirer.js не существует свойства конфигурации с именем `badProperty`. Выполните команду из листинга 2.37 в папке `todo` для компиляции кода в проекте.

Листинг 2.37. Запуск компилятора

`tsc`

Компилятор использует информацию о типе из листинга 2.35 и сообщает о следующей ошибке:

```
src/index.ts:25:9 - error TS2769: No overload matches this call.
Overload 1 of 2, '(questions: QuestionCollection<any>, initialAnswers?: Partial<any>): Promise<any> & { ui: Prompt<any>; }',
gave the following error.
Type '"list"' is not assignable to type '"number"'.
Overload 2 of 2, '(questions: QuestionCollection<any>, initialAnswers?: Partial<any>): Promise<any>', gave the following error.
Type '"list"' is not assignable to type '"number"'.

25      type: "list",
~~~~~
Found 1 error in src/index.ts:25
```

Декларация типа позволяет TypeScript предоставлять одинаковый набор функций в рамках всего приложения, даже несмотря на то, что пакет Inquirer.js представляет собой чистый JavaScript. Однако, как видно на этом примере, у такой возможности могут быть ограничения, и добавление свойства, которое не поддерживается, приводит к ошибке, связанной с присвоенным свойству type значением. Подобное происходит потому, что бывает трудно описать типы, которые ожидает чистый JavaScript, и иногда сообщения об ошибках могут быть скорее общим указанием на наличие какой-то проблемы.

2.4. ДОБАВЛЕНИЕ КОМАНД

Наше приложение на данный момент мало что делает и требует дополнительных команд. В последующих разделах мы добавим ряд новых команд и приведем реализацию каждой из них.

2.4.1. Фильтрация элементов

Первая команда, которую мы добавим, позволит пользователю переключать фильтр на отображение или скрытие завершенных дел, как показано в листинге 2.38.

Листинг 2.38. Фильтрация элементов в файле index.ts из папки src

```
import { TodoItem } from "./todoItem.js";
import { TodoCollection } from "./todoCollection.js";
import inquirer from "inquirer";

let todos: TodoItem[] = [
  new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
  new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
  console.log(`${collection.userName}'s Todo List '
    + `(${collection.getItemCounts().incomplete} items to do)');
  //collection.getTodoItems(true).forEach(item => item.printDetails());
  collection.getTodoItems(showCompleted)
    .forEach(item => item.printDetails());

}

enum Commands {
  Toggle = "Show/Hide Completed",
  Quit = "Quit"
}

function promptUser(): void {
  console.clear();
```

```

displayTodoList();
inquirer.prompt({
    type: "list",
    name: "command",
    message: "Choose option",
    choices: Object.values(Command),
    //badProperty: true
}).then(answers => {
    switch (answers["command"]) {
        case Commands.Toggle:
            showCompleted = !showCompleted;
            promptUser();
            break;
    }
})
}

promptUser();

```

Процесс добавления команд заключается в определении нового значения для перечисления `Commands` и операторов, реагирующих на выбор команды. В данном случае новое значение — `Toggle`, и когда оно выбрано, значение переменной `showCompleted` изменяется таким образом, чтобы функция `displayTodoList` включала или исключала завершенные элементы. Выполните команды из листинга 2.39 в папке `todo` для компиляции и запуска кода.

Листинг 2.39. Компиляция и выполнение

```
tsc
node dist/index.js
```

Выберите опцию `Show/Hide Completed` и нажмите `Enter` для переключения фильтра отображения завершенных задач в списке, как показано на рис. 2.3.

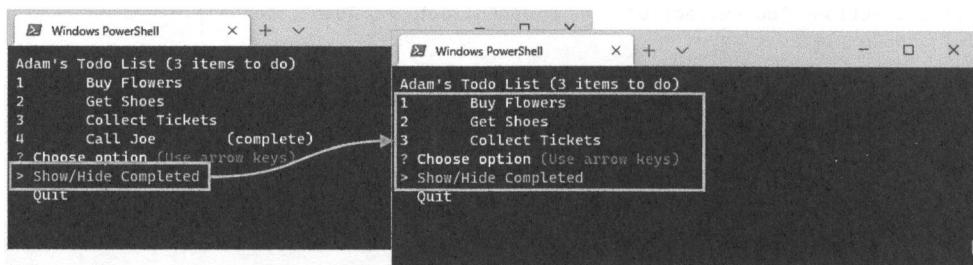


Рис. 2.3. Переключение фильтра отображения завершенных элементов

2.4.2. Добавление задач

Приложение не имеет большого смысла, если пользователь не сможет создавать новые задачи. В листинге 2.40 добавлена поддержка создания новых объектов `TodoItem`.

Листинг 2.40. Добавление задач в файл index.ts из папки src

```
import { TodoItem } from "./todoItem.js";
import { TodoCollection } from "./todoCollection.js";
import inquirer from "inquirer";

let todos: TodoItem[] = [
    new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
    new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
    console.log(`${collection.userName}'s Todo List ' +
        `+ ${collection.getItemCounts().incomplete} items to do)`);

    collection.getTodoItems(showCompleted)
        .forEach(item => item.printDetails());
}

enum Commands {
    Add = "Add New Task",
    Toggle = "Show/Hide Completed",
    Quit = "Quit"
}

function promptAdd(): void {
    console.clear();
    inquirer.prompt({ type: "input", name: "add", message: "Enter task:" })
        .then(answers => {if (answers["add"] !== "") {
            collection.addTodo(answers["add"]);
        }
        promptUser();
    })
}

function promptUser(): void {
    console.clear();
    displayTodoList();
    inquirer.prompt({
        type: "list",
        name: "command",
        message: "Choose option",
        choices: Object.values(Commands),
    }).then(answers => {
        switch (answers["command"]) {
            case Commands.Toggle:
                showCompleted = !showCompleted;
                promptUser();
                break;
            case Commands.Add:
                promptAdd();
                break;
        }
    })
}

promptUser();
```

Пакет Inquirer.js может предоставлять пользователю различные типы вопросов. Когда пользователь выбирает команду Add, тип входного вопроса `input` применяется для получения задачи, которая используется в качестве аргумента метода `TodoCollection.addTodo`. Для компиляции и выполнения кода выполните в папке `todo` команды из листинга 2.41.

Листинг 2.41. Компиляция и выполнение

```
tsc
node dist/index.js
```

Выберите опцию `Add New Task`, введите текст и нажмите `Enter`, чтобы создать новую задачу, как показано на рис. 2.4.

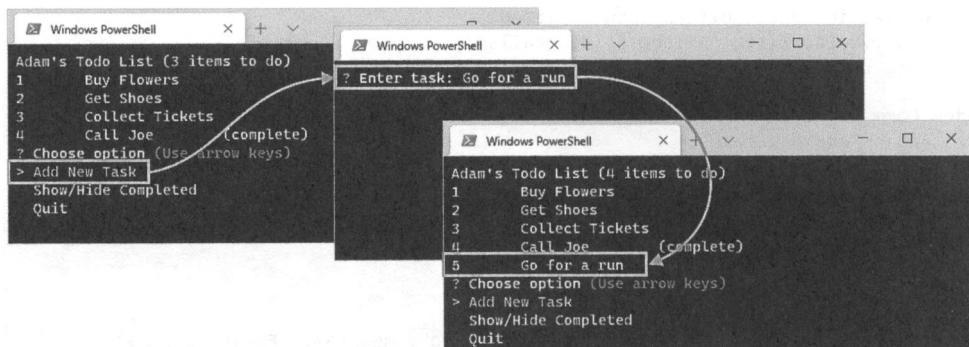


Рис. 2.4. Добавление новой задачи

2.4.3. Отметка о выполнении задачи

Завершение задачи — это двухэтапный процесс. Он требует от пользователя выбрать элемент, который необходимо пометить как завершенный. В листинге 2.42 добавлены команды и дополнительный запрос, что позволит пользователю отмечать задачи как выполненные и удалять их из списка дел.

Листинг 2.42. Завершение задач из списка дел в файле index.ts из папки src

```
import { TodoItem } from "./todoItem.js";
import { TodoCollection } from "./todoCollection.js";
import inquirer from "inquirer";

let todos: TodoItem[] = [
    new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
    new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
    console.log(`${collection.userName}'s Todo List '
        + `${collection.getItemCounts().incomplete } items to do)`);
```

```
collection.getTodoItems(showCompleted)
    .forEach(item => item.printDetails());
}

enum Commands {
    Add = "Add New Task",
    Complete = "Complete Task",

    Toggle = "Show/Hide Completed",
    Purge = "Remove Completed Tasks",

    Quit = "Quit"
}

function promptAdd(): void {
    console.clear();
    inquirer.prompt({ type: "input", name: "add", message: "Enter task:" })
        .then(answers => {if (answers["add"] !== "") {
            collection.addTodo(answers["add"]);
        }
        promptUser();
    })
}

function promptComplete(): void {
    console.clear();
    inquirer.prompt({ type: "checkbox", name: "complete",
        message: "Mark Tasks Complete",
        choices: collection.getTodoItems(showCompleted).map(item =>
            ({name: item.task, value: item.id, checked: item.complete}))
    }).then(answers => {
        let completedTasks = answers["complete"] as number[];
        collection.getTodoItems(true).forEach(item =>
            collection.markComplete(item.id,
                completedTasks.find(id => id === item.id) != undefined));
        promptUser();
    })
}

function promptUser(): void {
    console.clear();
    displayTodoList();
    inquirer.prompt({
        type: "list",
        name: "command",
        message: "Choose option",
        choices: Object.values(Commands),
    }).then(answers => {
        switch (answers["command"]) {
            case Commands.Toggle:
                showCompleted = !showCompleted;
                promptUser();
                break;
            case Commands.Add:
                promptAdd();
                break;
        }
    })
}
```

```

        case Commands.Complete:
            if (collection.getItemCounts().incomplete > 0) {
                promptComplete();
            } else {
                promptUser();
            }
            break;
        case Commands.Purge:
            collection.removeComplete();
            promptUser();
            break;
    }
}
}

promptUser();

```

Эти изменения приводят к добавлению в приложение новой команды, которая предоставляет пользователю список задач и позволяет изменять их состояние. Переменная `showCompleted` используется для определения того, будут ли отображаться завершенные элементы, что создает связь между командами `Toggle` и `Complete`.

Единственное новое свойство TypeScript, которое можно отметить, содержится в этом выражении:

```

...
let completedTasks = answers["complete"] as number[];
...

```

Даже при наличии деклараций типов бывают случаи, когда TypeScript не способен правильно определить используемые типы. В таком случае пакет `Inquirer.js` позволяет использовать любой тип данных в подсказках, отображаемых пользователю, и компилятор не может определить, что использовано только значение типа `number`, а значит, и в качестве ответов могут быть получены значения типа `number`. Для решения этой проблемы мы используем *утверждение типа*, которое дает возможность указать компилятору использовать заданный тип, даже если он определил другой тип данных (или вообще не определил его). Когда используется утверждение типа, оно переопределяет действия компилятора, а значит, разработчик несет ответственность за правильность указанного типа. Для компиляции и выполнения кода выполните команды из листинга 2.43 в папке `todo`.

Листинг 2.43. Компиляция и выполнение

```
tsc
node dist/index.js
```

Выберите опцию `Complete Task`, с помощью пробела выделите одну или несколько задач для изменения, а затем нажмите `Enter`. Состояние выбранных элементов будет изменено, что отразится в обновленном списке, как показано на рис. 2.5.

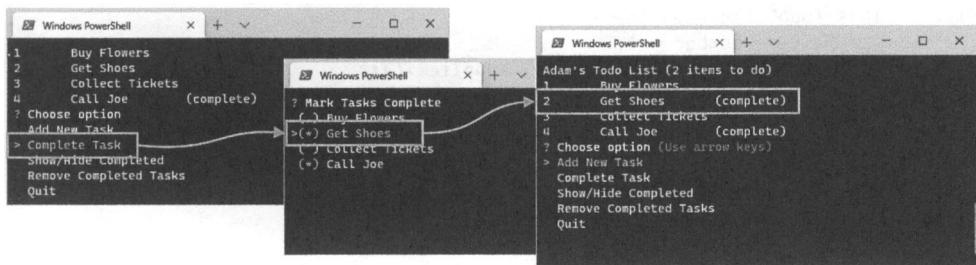


Рис. 2.5. Завершение задач

2.5. ПОСТОЯННОЕ ХРАНЕНИЕ ДАННЫХ

Для постоянного хранения элементов списка дел мы выберем другой пакет с открытым исходным кодом, поскольку нет смысла создавать новую функциональность, когда существуют хорошо написанные и хорошо протестированные альтернативы. Выполните команды из листинга 2.44 в папке `todo` для установки пакета `lowdb` и определений типов, описывающих его API в TypeScript.

Листинг 2.44. Добавление пакета

```
npm install lowdb@5.1.0
```

`Lowdb` – это отличный инструмент для работы с базами данных, который хранит данные в JSON-файле и используется в качестве компонента хранения данных в пакете `json-server` при создании HTTP-веб-сервисов в части III.

Обратите внимание, что мы не устанавливали никаких деклараций типов для этого пакета. TypeScript стал настолько популярным, что многие пакеты, включая `Lowdb`, поставляются с декларациями типов как часть пакета JavaScript.

СОВЕТ

Я не стал детально описывать API `Lowdb` в книге, поскольку он не имеет прямого отношения к TypeScript. Если вы хотите использовать `Lowdb` в своих проектах, для получения подробной информации о пакете перейдите по ссылке <https://github.com/typicode/lowdb>.

Давайте реализуем постоянное хранилище данных, применяя наследование класса `TodoCollection`. В процессе подготовки изменим ключевое слово управления доступом, используемое классом `TodoCollection`, чтобы подклассы могли получить доступ к `Map`, содержащей объекты `TodoItem`, как показано в листинге 2.45.

Листинг 2.45. Изменение управления доступом в файле `todoCollection.ts` из папки `src`

```

import { TodoItem } from "./todoItem.js";

type ItemCounts = {
  total: number,
  incomplete: number
}

```

```

export class TodoCollection {
    private nextId: number = 1;
    protected itemMap = new Map<number, TodoItem>();

    constructor(public userName: string, todoItems: TodoItem[] = []) {
        todoItems.forEach(item => this.itemMap.set(item.id, item));
    }

    // ...методы для краткости опущены...
}

```

Ключевое слово `protected` указывает компилятору TypeScript на то, что доступ к свойству может быть предоставлен только классу или его подклассам. Для создания подкласса добавьте в папку `src` файл `jsonTodoCollection.ts` с кодом, показанным в листинге 2.46.

Листинг 2.46. Содержимое файла `jsonTodoCollection.ts` из папки `src`

```

import { TodoItem } from "./todoItem.js";
import { TodoCollection } from "./todoCollection.js";
import { LowSync } from "lowdb";
import { JSONFileSync } from "lowdb/node";

type schemaType = {
    tasks: { id: number; task: string; complete: boolean; }[]
};

export class JsonTodoCollection extends TodoCollection {
    private database: LowSync<schemaType>;

    constructor(public userName: string, todoItems: TodoItem[] = []) {
        super(userName, []);
        this.database = new LowSync(new JSONFileSync("Todos.json"));
        this.database.read();

        if (this.database.data == null) {
            this.database.data = { tasks: todoItems };
            this.database.write();
            todoItems.forEach(item => this.itemMap.set(item.id, item));
        } else {
            this.database.data.tasks.forEach(item =>
                this.itemMap.set(item.id,
                    new TodoItem(item.id, item.task, item.complete)));
        }
    }

    addTodo(task: string): number {
        let result = super.addTodo(task);
        this.storeTasks();
        return result;
    }

    markComplete(id: number, complete: boolean): void {
        super.markComplete(id, complete);
        this.storeTasks();
    }
}

```

```

removeComplete(): void {
    super.removeComplete();
    this.storeTasks();
}

private storeTasks() {
    this.database.data.tasks = [...this.itemMap.values()];
    this.database.write();
}
}

```

Определение типа для Lowdb включает в себя схему для описания структуры хранимых данных, которая затем применяется с аргументами обобщенного типа, чтобы компилятор смог проверить используемые типы данных. Например, если нам необходимо хранить только один тип данных, описываемый с помощью псевдонима типа.

```

...
type schemaType = {
    tasks: { id: number; task: string; complete: boolean; }[]
};

...

```

Тип схемы используется при создании базы данных Lowdb, и компилятор проверяет, как используются данные при их чтении из базы, как, например, в этом выражении:

```

...
this.database.data.tasks.forEach(item => this.itemMap.set(item.id,
    new TodoItem(item.id, item.task, item.complete)));
...

```

Компилятор знает, что свойство `tasks`, представленное данными, соответствует свойству `tasks` в типе схемы, и возвращает массив объектов со свойствами `id`, `task` и `complete`.

В файле `index.ts` (листинг 2.47) используется класс `JsonTodoCollection`, поэтому данные будут храниться в приложении на постоянной основе.

Листинг 2.47. Практическое применение постоянной коллекции в файле `index.ts` из папки `src`

```

import { TodoItem } from "./todoItem.js";
import { TodoCollection } from "./todoCollection.js";
import inquirer from "inquirer";
import { JsonTodoCollection } from "./jsonTodoCollection.js";

let todos: TodoItem[] = [
    new TodoItem(1, "Buy Flowers"), new TodoItem(2, "Get Shoes"),
    new TodoItem(3, "Collect Tickets"), new TodoItem(4, "Call Joe", true)];

let collection: TodoCollection = new JsonTodoCollection("Adam", todos);

let showCompleted = true;

...

```

Наконец, выполните команды из листинга 2.48 в папке `todo`, чтобы скомпилировать и выполнить код в последний раз в этой главе.

Листинг 2.48. Компиляция и выполнение

```
tsc  
node dist/index.js
```

При запуске приложения в каталоге `todo` будет создан файл `Todos.json`, используемый для хранения JSON-представления объектов `TodoItem`. Это гарантирует, что ни одно изменение не будет потеряно при завершении работы приложения.

РЕЗЮМЕ

В этой главе был создан простой пример приложения для демонстрации разработки на TypeScript и некоторых основных концепций данного языка. Мы рассмотрели возможности TypeScript, которые расширяют JavaScript, фокусируются на безопасности типов и помогают избежать распространенных шаблонов, которые сбивают с толку многих разработчиков, особенно тех, кто до JavaScript работал на таких языках, как C# или Java.

Вы видели, что TypeScript не используется изолированно. Для выполнения JavaScript-кода, создаваемого компилятором TypeScript, требуется среда выполнения JavaScript. Преимущество такого подхода заключается в том, что проекты, написанные с использованием TypeScript, имеют полный доступ к широкому спектру пакетов JavaScript, многие из которых имеют определения типов, доступные для удобного использования.

- Разработка TypeScript может выполняться с помощью свободно распространяемых инструментов.
- TypeScript базируется на языке JavaScript, а его главная особенность – статическая типизация.
- Результатом работы компилятора TypeScript является чистый JavaScript, который может быть выполнен под подходящей средой выполнения JavaScript.
- Приложения на TypeScript могут использовать стандартные пакеты JavaScript, хотя для подготовки проекта TypeScript перед их установкой могут потребоваться базовые знания модулей JavaScript.
- Некоторые пакеты JavaScript включают информацию о типах для использования с TypeScript.
- Существуют отдельные пакеты, предназначенные для добавления деклараций типов к популярным пакетам, которые сами по себе не содержат таких деклараций.

Приложение, которое было создано в этой главе, демонстрирует некоторые ключевые возможности TypeScript, однако их гораздо больше, как можно судить по объему книги. В следующей главе мы рассмотрим, как TypeScript вписывается в общую картину разработки, а также поговорим о структуре книги и ее содержании.

3

Обзор JavaScript, часть 1

В этой главе

- ✓ Использование типов JavaScript.
- ✓ Приведение типов JavaScript.
- ✓ Определение и использование функций и массивов JavaScript.
- ✓ Создание и реализация объектов JavaScript.
- ✓ Особенности ключевого слова `this`.

Эффективная разработка на TypeScript требует понимания того, как JavaScript работает с типами данных. Это может вызвать недоумение у разработчиков, переходящих на TypeScript из-за сложностей, с которыми они сталкивались в JavaScript. Однако понимание JavaScript упрощает изучение TypeScript и предоставляет ценные знания о том, что предлагает TypeScript и как работают его функции. В этой главе вы познакомитесь с основными возможностями типов в JavaScript, а в главе 4 мы перейдем к более продвинутым возможностям.

3.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

Прежде чем начать работать с главой, создайте в удобном месте папку `primer`. Откройте командную строку, перейдите в папку `primer` и выполните команду из листинга 3.1.

Листинг 3.1. Подготовка папки проекта

```
npm init --yes
```

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Для установки пакета, который будет автоматически запускать JavaScript-файл при изменении его содержимого, выполните в папке `primer` команду, показанную в листинге 3.2.

Листинг 3.2. Установка пакета

```
npm install nodemon@2.0.20
```

Пакет под названием `nodemon` загрузится и установится. Затем создайте в папке `primer` файл `index.js` с содержимым, показанным в листинге 3.3.

Листинг 3.3. Содержимое файла index.js из папки primer

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
```

Наберите команду из листинга 3.4, чтобы запустить JavaScript-файл и отслеживать его изменения.

Листинг 3.4. Запуск мониторинга файлов JavaScript

```
npx nodemon index.js
```

Пакет `nodemon` выполнит содержимое файла `index.js` и выдаст следующий результат:

```
[nodemon] 2.0.20
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
Hat price: 100
[nodemon] clean exit - waiting for changes before restart
```

Здесь выделена та часть вывода, которая получена из файла `index.js`. Чтобы убедиться в правильности отслеживания изменений, измените содержимое файла `index.js`, как показано в листинге 3.5.

Листинг 3.5. Внесение изменений в файл index.js из папки primer

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');
```

После сохранения изменений пакет `nodemon` должен обнаружить, что файл `index.js` был модифицирован, и выполнить содержащийся в нем код. Код в листинге 3.5 выдает следующий результат, который показан без учета информации, предоставляемой пакетом `nodemon`:

```
Hat price: 100
Boots price: 100
```

3.2. ЗАПУТАННЫЙ JAVASCRIPT

JavaScript обладает множеством особенностей, которые делают его схожим с другими языками программирования. Обычно разработчики начинают с кода, аналогичного приведенному в листинге 3.5. Даже если вы новичок в JavaScript, выражения в листинге 3.5 покажутся вам знакомыми.

Основными строительными блоками кода JavaScript являются операторы, которые выполняются в том порядке, в котором они написаны в коде. Для объявления переменных предназначено ключевое слово `let` (в отличие от `const`, используемого для создания постоянных значений), за которым следует имя переменной. Значение переменной задается с помощью оператора присваивания (знак равенства), за которым следует само значение.

JavaScript предоставляет некоторые встроенные объекты для решения распространенных задач, таких как вывод строк в консоль с помощью метода `console.log`. Строки можно задавать как литеральные значения, заключая их в одинарные или двойные кавычки, или как строки-шаблоны, используя обратные кавычки и вставляя в них выражения с помощью символа доллара и фигурных скобок.

Тем не менее иногда возникают неожиданные результаты. Причина путаницы заключается в том, как JavaScript работает с типами. В листинге 3.6 показан типичный пример подобной проблемы.

Листинг 3.6. Добавление операторов в файл index.ts

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

if (hatPrice == bootsPrice) {
    console.log("Prices are the same");
} else {
    console.log("Prices are different");
}

let totalPrice = hatPrice + bootsPrice;
console.log('Total Price: ${totalPrice}'');
```

Новые операторы сравнивают значения переменных `hatPrice` и `bootsPrice`, а затем присваивают их сумму новой переменной `totalPrice`. Метод `console.log` используется для вывода сообщений в командную строку, и при выполнении кода получается следующий результат:

```
Hat price: 100
Boots price: 100
Prices are the same
Total Price: 100100
```

Большинство разработчиков заметят, что значение переменной `hatPrice` представлено числом, а `bootsPrice` — строкой, заключенной в двойные кавычки.

В большинстве языков программирования операции над разными типами данных вызывают ошибку. В JavaScript все иначе: сравнение строки и числа проходит успешно, а попытка суммировать значения приводит к их конкатенации. Понимание результатов из листинга 3.6 и причин, лежащих в их основе, помогает раскрыть детали подхода JavaScript к типам данных и объясняет, почему TypeScript может быть так полезен.

3.3. ТИПЫ ДАННЫХ В JAVASCRIPT

Может показаться, что в JavaScript нет типов данных или что они используются непоследовательно, но это не так. Просто JavaScript работает иначе, чем большинство популярных языков программирования, и его поведение кажется нелогичным только до тех пор, пока вы не поймете, чего от него ожидать. Основой JavaScript является набор встроенных типов, которые описаны в табл. 3.1.

Таблица 3.1. Встроенные типы JavaScript

Название	Описание
number	Используется для представления числовых значений. В отличие от других языков программирования, в JavaScript нет разделения на целочисленные значения и значения с плавающей точкой, поэтому оба они могут быть представлены с помощью этого типа
string	Используется для представления текстовых данных
boolean	Логический тип данных, который может иметь два возможных значения — <code>true</code> и <code>false</code>
symbol	Используется для представления уникальных постоянных значений, таких как ключи в коллекциях
null	Этому типу может быть присвоено только значение <code>null</code> . Применяется для обозначения несуществующей или недопустимой ссылки
undefined	Данный тип используется, когда переменная была объявлена, но ей не было присвоено значение
object	Применяется для представления составных значений, образованных из отдельных свойств и значений

Первые шесть типов — это примитивные типы данных JavaScript. Они всегда доступны, и каждое значение в приложении JavaScript либо само является примитивным типом, либо состоит из них. Седьмой — `object` — используется для представления объектов.

3.3.1. Работа с примитивными типами данных

Если вернуться к листингу 3.6, можно заметить, что в коде не указан ни один конкретный тип данных. В других языках программирования требуется объявление типа переменной перед ее использованием, как, например, в этом фрагменте кода из одной из моих книг по C#:

```
...
string name = "Adam";
...
```

Этот оператор определяет, что тип переменной `name` — строка (`string`), и присваивает ей значение `Adam`. В JavaScript типы имеют *значения*, а не переменные. Чтобы определить переменную, содержащую строку, нужно присвоить ей строковое значение, как показано в листинге 3.7.

Листинг 3.7. Создание строковой переменной в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

if (hatPrice == bootsPrice) {
    console.log("Prices are the same");
} else {
    console.log("Prices are different");
}

let totalPrice = hatPrice + bootsPrice;
console.log('Total Price: ${totalPrice}');

let myVariable = "Adam";
```

Среда выполнения JavaScript должна только определить, какой из типов из табл. 3.1 ей следует использовать для значения, присвоенного переменной `myVariable`. Небольшой набор поддерживаемых JavaScript типов упрощает этот процесс. Среда выполнения знает, что любое значение, заключенное в двойные кавычки, должно быть строкой. Уточнить тип значения можно с помощью ключевого слова `typeof` (листинг 3.8).

Листинг 3.8. Определение типа значения в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

if (hatPrice == bootsPrice) {
    console.log("Prices are the same");
} else {
```

```

        console.log("Prices are different");
    }

let totalPrice = hatPrice + bootsPrice;
console.log('Total Price: ${totalPrice}');

let myVariable = "Adam";
console.log('Type: ${typeof myVariable}');

```

Ключевое слово `typeof` определяет тип значения и при выполнении кода выдает следующий результат:

```

Hat price: 100
Boots price: 100
Prices are the same
Total Price: 100100
Type: string

```

В листинге 3.9 переменной `myVariable` присваивается новое значение и снова выводится ее тип.

Листинг 3.9. Присвоение нового значения в файле index.js

```

let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

if (hatPrice == bootsPrice) {
    console.log("Prices are the same");
} else {
    console.log("Prices are different");
}

let totalPrice = hatPrice + bootsPrice;
console.log('Total Price: ${totalPrice}');

let myVariable = "Adam";
console.log('Type: ${typeof myVariable}');
myVariable = 100;
console.log('Type: ${typeof myVariable}');

```

После сохранения изменений код выдаст следующий результат:

```

Hat price: 100
Boots price: 100
Prices are the same
Total Price: 100100
Type: string
Type: number

```

Изменение значения, присвоенного переменной, приводит к изменению типа, который возвращает ключевое слово `typeof`, поскольку значения имеют свои типы. Изначально переменной `myVariable` было присвоено значение типа `string`, а затем — значение типа `number`. Такой динамический характер типизации облегчается ограниченным набором типов, поддерживаемых JavaScript, что упрощает

определение используемого встроенного типа. Например, если все числа представлены типом `number`, это означает, что и целые, и числа с плавающей точкой обрабатываются с помощью `number`. Это было бы невозможно при использовании более сложной системы типов.

ОСОБЕННОСТИ TYPEOF NULL

Когда ключевое слово `typeof` используется для значений `null`, результатом является `object`. Это поведение было известным с самых ранних этапов развития JavaScript и осталось неизменным. Множество существующего кода предполагает такое поведение, поэтому его не меняли.

3.3.2. Приведение типа

Когда оператор применяется к значениям разных типов, среда выполнения JavaScript автоматически преобразует одно из них в эквивалентное ему значение другого типа. Данный процесс называется *приведением типов* или *преобразованием типов*. Именно это служит причиной того, что результаты в листинге 3.6 кажутся противоречивыми. Хотя, как вы узнаете далее, результаты на самом деле не являются таковыми, если понять, как работает такой механизм. В листинге 3.6 есть две точки в коде, где происходит приведение типов.

```
...
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

if (hatPrice == bootsPrice) {
...
}
```

Двойной знак равенства выполняет сравнение с одновременным использованием приведения типов. Это значит, что JavaScript пытается преобразовать сравниваемые значения для получения смыслового результата. Данный процесс известен как *абстрактное сравнение равенства* JavaScript, и когда тип `number` сравнивается со `string`, значение `string` преобразуется в `number`, а затем выполняется сравнение. Таким образом, если числовое и строковое значения `100` сравниваются между собой, строка будет преобразована в число и выражение `if` вернет значение `true`.

СОВЕТ

Если вам интересно, какую последовательность шагов выполняет JavaScript при абстрактном сравнении равенств, обратитесь к спецификации JavaScript (<https://262.ecma-international.org/13.0/#sec-islooselyequal>). Спецификация хорошо написана и довольно интересна. Однако, прежде чем потратить целый день на изучение деталей реализации, помните, что TypeScript имеет ограничения на использование некоторых самых необычных и экзотических опций.

Второе использование приведения типов в листинге 3.6 происходит при суммировании цен.

```
...
let totalPrice = hatPrice + bootsPrice;
...
```

При использовании оператора `+` для типов `number` и `string` одно из значений преобразуется для соответствия другому типу. Однако такое преобразование отличается от того, что происходит при сравнении. Если одно из значений — `string`, то второе также преобразуется в `string`, и оба уже строковых значения конкatenируются. То есть `number` и `string` объединяются в `string`, а их значения просто «склеиваются» друг с другом. В нашем случае сложение (оператор `+`) числа `100` и строки `"100"` привело к строковому результату `100100`.

Предотвращение непреднамеренного приведения типа

Приведение типов может быть полезной функцией, но она приобрела плохую репутацию только потому, что используется непреднамеренно. Это легко сделать, когда обрабатываемые типы меняются на новые значения. Как вы узнаете из последующих глав, в TypeScript предусмотрены функции, которые помогают управлять нежелательным приведением типов. Но JavaScript также предоставляет возможности для предотвращения приведения, как показано в листинге 3.10.

Листинг 3.10. Предотвращение приведения в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

if (hatPrice === bootsPrice) {
    console.log("Prices are the same");
} else {
    console.log("Prices are different");
}

let totalPrice = Number(hatPrice) + Number(bootsPrice);
console.log('Total Price: ${totalPrice}');
let myVariable = "Adam";
console.log('Type: ${typeof myVariable}');
myVariable = 100;
console.log('Type: ${typeof myVariable}'');
```

Двойной знак равенства (`==`) выполняет сравнение с применением приведения типа. Тройной знак равенства (`====`) выполняет строгое сравнение, возвращая `true` только в том случае, если значения имеют одинаковый тип и равны.

Чтобы избежать конкатенации строк, значения перед применением оператора `+` могут быть явно преобразованы в числа с помощью встроенной функции `Number`, в результате чего выполняется сложение чисел. Код, приведенный в листинге 3.10, выдает следующий результат:

```
Hat price: 100
Boots price: 100
Prices are different
Total Price: 200
Type: string
Type: number
```

Оценка эффективности явно применяемого приведения типа

Принудительное приведение типов может быть полезным, когда используется в явном виде. Например, одной из полезных функций является способ приведения значений к типу `boolean` с помощью логического оператора ИЛИ (`||`). Значения типа `null` или `undefined` преобразуются в `false`, что является эффективным инструментом предоставления резервных значений, как показано в листинге 3.11.

Листинг 3.11. Обработка значений типа `null` в файле `index.js`

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

if (hatPrice === bootsPrice) {
    console.log("Prices are the same");
} else {
    console.log("Prices are different");
}

let totalPrice = Number(hatPrice) + Number(bootsPrice);
console.log('Total Price: ${totalPrice}');

let myVariable = "Adam";
console.log('Type: ${typeof myVariable}');
myVariable = 100;
console.log('Type: ${typeof myVariable}');

let firstCity;
let secondCity = firstCity || "London";
console.log('City: ${secondCity }');
```

Значение переменной `secondCity` задается с помощью выражения, проверяющего значение `firstCity`: если `firstCity` преобразуется в логическое значение `true`, то `secondCity` получит то же значение, что и `firstCity`.

Тип `undefined` используется, когда переменные определены, но им не присвоено значение, как в случае с переменной `firstCity`. Использование оператора `||` гарантирует, что резервное значение `secondCity` будет использоваться, если `firstCity` не определено (`undefined`) или равно `null`.

Оператор нулевого слияния

Одна из проблем логического оператора ИЛИ заключается в том, что в значение `false` преобразуется не только `null` или `undefined`. Это может привести к неожиданным результатам, как показано в листинге 3.12.

Листинг 3.12. Эффект приведения типа в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

let taxRate; // ставка налога не задана
console.log('Tax rate: ${taxRate || 10}%');
taxRate = 0; // ставка налога равна нулю
console.log('Tax rate: ${taxRate || 10}%');
```

Помимо `null` и `undefined`, логический оператор ИЛИ приводит к значению `false` число `0`, пустую строку `("")` и специальное числовое значение `NaN`. Эти значения, а также значение `false` в JavaScript называются ложными и могут вас запутать. В листинге 3.12 логический оператор ИЛИ использует резервное значение, когда переменной `taxRate` присваивается нулевое значение, и выдает следующий результат:

```
Hat price: 100
Boots price: 100
Tax rate: 10%
Tax rate: 10%
```

Код не различает неприсвоенное и нулевое значение, что может стать проблемой, когда ноль является обязательным значением. В данном примере невозможно установить налоговую ставку равной нулю, хотя это вполне допустимая ставка. Для решения данной проблемы в JavaScript есть оператор нулевого слияния (Nullish Coalescing) — `??`, который объединяет только значения `undefined` и `null`, но не другие ложные значения (листинг 3.13).

Листинг 3.13. Практическое применение оператора нулевого слияния в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

let taxRate; // ставка налога не задана
console.log('Tax rate: ${taxRate ?? 10}%');

taxRate = 0; // ставка налога равна нулю
console.log('Tax rate: ${taxRate ?? 10}%');
```

В первом операторе будет использовано резервное значение, поскольку значение `taxRate` — `undefined`. Во втором операторе резервное значение не будет использовано, поскольку для нуля не выполняется приведение типа оператором `??`, и это даст следующий результат:

```
Hat price: 100
Boots price: 100
Tax rate: 10%
Tax rate: 0%
```

3.3.3. Работа с функциями

Гибкий подход JavaScript к типам используется и в других частях языка, включая функции. В листинге 3.14 в файл JavaScript добавлена функция и для краткости удалены некоторые операторы из предыдущих примеров.

Листинг 3.14. Определение функции в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

function sumPrices(first, second, third) {
    return first + second + third;
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log('Total Price: ${totalPrice}');
```

Типы параметров функции определяются значениями, которые передаются при ее вызове. Например, функция может ожидать, что ей будут переданы значения типа `number`. Однако ничто не мешает вызвать функцию с аргументами других типов, такими как `string`, `boolean` или `object`. Непредвиденные результаты могут возникнуть, если функция не учитывает эти варианты, либо из-за принудительного преобразования типов средой выполнения JavaScript, либо из-за особенностей, свойственных определенному типу.

Функция `sumPrices` из листинга 3.14 использует оператор `+`, предназначенный для суммирования набора числовых параметров. Однако одно из значений, переданных при вызове функции, является строкой. Как уже объяснялось ранее, оператор `+`, примененный к строке, выполняет конкатенацию, поэтому код из листинга 3.14 выдает следующий результат:

```
Hat price: 100
Boots price: 100
Total Price: 100100undefined
```

В JavaScript не обеспечивается соответствие между количеством параметров, определяемых функцией, и количеством аргументов, используемых при ее вызове. Любой параметр, для которого не указано значение, будет автоматически считаться `undefined`. В листинге 3.14 у параметра с именем `third` нет значения, поэтому ему было присвоено `undefined`. Затем оно преобразуется в строку и включается в вывод конкатенации:

```
Total Price: 100100undefined
```

Работа с результатами функций

Различия между типами в JavaScript и в других языках программирования увеличиваются при работе с функциями. Следствием особенностей типов JavaScript является то, что аргументы, используемые при вызове функции, могут влиять на тип возвращаемого функцией значения, как показано в листинге 3.15.

Листинг 3.15. Вызов функции в файле index.js

```

let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

function sumPrices(first, second, third) {
    return first + second + third;
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, 300);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

```

Значение переменной `totalPrice` устанавливается три раза путем вызова функции `sumPrices`. После каждого вызова функции используется ключевое слово `typeof` для определения типа значения, возвращаемого функцией. Код из листинга 3.15 выдает следующий результат:

```

Hat price: 100
Boots price: 100
Total: 100100undefined string
Total: 600 number
Total: NaN number

```

Первый вызов функции включает строковый аргумент, в результате чего все параметры функции преобразуются в строковые значения и конкатенируются, то есть функция возвращает строковое значение `100100undefined`.

Второй вызов использует три числовых значения, которые складываются и дают числовой результат `600`.

Последний вызов функции также использует числовые аргументы. Однако, поскольку третий аргумент не указан, JavaScript обрабатывает его параметр как `undefined`. JavaScript выполняет приведение `undefined` параметра в специальное **числовое значение `Nan`** (not a number, или «не число»). Результатом операции сложения, включающей `Nan`, является `Nan`, то есть тип результата — `number`, но его значение не является полезным и вряд ли может быть использовано в большинстве случаев.

Предотвращение проблем, связанных с несоответствием аргументов

Хотя результаты, приведенные в предыдущем подразделе, могут сбить с толку, они в точности соответствуют спецификации JavaScript. Проблема не в том, что JavaScript непредсказуем, а в том, что его подход отличается от подходов других популярных языков программирования.

JavaScript предоставляет возможности, позволяющие избежать подобных проблем. Одна из них — это значения по умолчанию для параметров, которые используются, если функция вызывается без соответствующего аргумента. Пример такого использования представлен в листинге 3.16.

Листинг 3.16. Использование значения параметра по умолчанию в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

function sumPrices(first, second, third = 0) {
    return first + second + third;
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, 300);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200);
console.log('Total: ${totalPrice} ${typeof totalPrice}');
```

За именем параметра `third` следует знак равенства и значение, которое будет использовано, если функция вызывается без соответствующего значения. В результате оператор, вызывающий функцию `sumPrices` с двумя числовыми значениями, больше не будет выдавать результат `Nan`:

```
Hat price: 100
Boots price: 100
Total: 100100 string
Total: 600 number
Total: 300 number
```

Более гибкий подход — использовать `rest`-параметр: ему предшествуют три точки (...), и он должен быть последним параметром, определяемым функцией, как показано в листинге 3.17.

Листинг 3.17. Использование параметра `rest` в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

function sumPrices(...numbers) {
    return numbers.reduce(function(total, val) {
        return total + val
    }, 0);
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, 300);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200);
console.log('Total: ${totalPrice} ${typeof totalPrice}');
```

Параметр `rest` – это массив, содержащий все аргументы, для которых параметры не были определены. В функции из листинга 3.17 определен только параметр `rest`, а значит, его значением будет массив, содержащий все аргументы, использованные при вызове функции. Содержимое массива суммируется с помощью встроенного метода `reduce`. Массивы JavaScript описаны в разделе 3.4, а метод `reduce` используется для вызова функции для каждого элемента массива с получением одного значения результата. Такой подход гарантирует, что количество аргументов не влияет на результат, но функция, вызываемая методом `reduce`, использует оператор сложения (`+`), а это означает, что строковые значения все равно будут конкатенированы. Приведенный выше листинг выдает следующий результат:

```
Hat price: 100
Boots price: 100
Total: 100100 string
Total: 600 number
Total: 300 number
```

Для того чтобы функция возвращала верный результат суммы значений своих параметров, какими бы они ни были, параметры можно преобразовать в числа и удалить те, которые являются `NaN`, как показано в листинге 3.18.

Листинг 3.18. Преобразование и фильтрация параметров в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice}');

function sumPrices(...numbers) {
    return numbers.reduce(function(total, val) {
        return total + (Number.isNaN(Number(val)) ? 0 : Number(val));
    }, 0);
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, 300);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, undefined, false, "hello");
console.log('Total: ${totalPrice} ${typeof totalPrice}');
```

Для проверки того, является ли числовое значение `NaN`, используется метод `Number.isNaN`. Код в листинге 3.18 явно преобразует каждый параметр в число и заменяет нулем те, которые имеют значение `NaN`. При этом обрабатываются только те значения параметров, которые можно рассматривать как числа, а аргументы типа `undefined`, `boolean` и `string`, добавленные в финальный вызов функции, не влияют на результат:

```
Hat price: 100
Boots price: 100
Total: 200 number
Total: 600 number
Total: 300 number
```

Использование стрелочных функций

Стрелочные функции, также известные как функции толстых стрелок, или *лямбда-выражения*, часто используются для определения функций, передаваемых в качестве аргументов в другие функции. В листинге 3.19 стандартная функция, используемая в методе `reduce`, заменена стрелочной функцией.

Листинг 3.19. Использование стрелочной функции в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice>');

function sumPrices(...numbers) {
    return numbers.reduce((total, val) =>
        total + (Number.isNaN(Number(val)) ? 0 : Number(val)));
}

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, 300);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, undefined, false, "hello");
console.log('Total: ${totalPrice} ${typeof totalPrice}');
```

Стрелочная функция состоит из трех частей: входных параметров, знака равенства с символом «больше» (образуя в совокупности стрелку `=>`) и, наконец, возвращаемого значения. Ключевое слово `return` и фигурные скобки нужны только в том случае, если в теле стрелочной функции содержится более одного оператора. Код из листинга 3.19 выдает тот же результат, что и код из листинга 3.18.

Стрелочные функции можно использовать везде, где требуется функция, и в целом их использование — это лишь вопрос личных предпочтений (но нужно иметь в виду проблему, описанную в разделе 3.6). В листинге 3.20 функция `sumPrices` переопределена с учетом стрелочного синтаксиса. Код из него приводит к тому же результату, что и код из листинга 3.18.

Листинг 3.20. Замена функции в файле index.js

```
let hatPrice = 100;
console.log('Hat price: ${hatPrice}');
let bootsPrice = "100";
console.log('Boots price: ${bootsPrice>');

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
    total + (Number.isNaN(Number(val)) ? 0 : Number(val)));

let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, 300);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

totalPrice = sumPrices(100, 200, undefined, false, "hello");
console.log('Total: ${totalPrice} ${typeof totalPrice}');
```

Функции — вне зависимости от используемого синтаксиса — тоже являются значениями. Они представляют собой особую категорию типа `object` (подробнее о котором написано в разделе 3.5). Функции могут присваиваться переменным, передаваемым в качестве аргументов другим функциям, и использоваться как любое другое значение.

В листинге 3.20 синтаксис стрелочной функции используется для определения функции, которой присваивается переменная `sumPrices`. Функции уникальны тем, что они могут быть вызваны, но возможность рассматривать их как значения позволяет лаконично выразить сложную функциональность. Однако при этом легко создать трудночитаемый код. В книге еще будут приведены примеры использования стрелочных функций и функций в качестве значений.

3.4. РАБОТА С МАССИВАМИ

Массивы в JavaScript соответствуют подходу, принятому в большинстве языков программирования. Отличие лишь в том, что они динамически изменяются по размеру и могут содержать любую комбинацию значений, включая любую комбинацию типов. В листинге 3.21 показано, как определяется и используется массив.

Листинг 3.21. Определение и использование массива в файле `index.js`

```
let names = ["Hat", "Boots", "Gloves"];
let prices = [];

prices.push(100);
prices.push("100");
prices.push(50.25);

console.log('First Item: ${names[0]}: ${prices[0]}');

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
  total + (Number.isNaN(Number(val)) ? 0 : Number(val)));

let totalPrice = sumPrices(...prices);
console.log('Total: ${totalPrice} ${typeof totalPrice}');
```

Размер массива при его создании не задается, а емкость распределяется автоматически по мере добавления или удаления элементов. Индексация элементов массива в JavaScript начинается с 0, а сам массив определяется с помощью квадратных скобок, при этом начальное содержимое его элементов разделяется запятыми. Например, массив `names` в листинге 3.21 был инициализирован с тремя строковыми значениями, а массив `prices` объявлен пустым. Для добавления элементов в конец массива используется метод `push`. Листинг выводит следующий результат:

```
First Item: Hat: 100
Total: 250.25 number
```

Элементы массива можно читать и устанавливать с помощью квадратных скобок. Кроме того, они могут быть обработаны с использованием методов, описанных в табл. 3.2.

Таблица 3.2. Полезные методы работы с массивами

Метод	Описание
<code>concat(otherArray)</code>	Возвращает новый массив, который объединяет массив, для которого он был вызван, с массивом, указанным в качестве аргумента. Можно указать несколько массивов
<code>join(separator)</code>	Объединяет все элементы массива в строку. Аргумент задает символ, используемый для разделения элементов
<code>pop()</code>	Удаляет и возвращает последний элемент массива
<code>shift()</code>	Удаляет и возвращает первый элемент массива
<code>push(item)</code>	Добавляет указанный элемент в конец массива
<code>unshift(item)</code>	Вставляет новый элемент в начало массива
<code>reverse()</code>	Возвращает новый массив, содержащий элементы в обратном порядке
<code>slice(start,end)</code>	Возвращает новый массив, содержащий элементы из указанного диапазона исходного массива
<code>sort()</code>	Сортирует элементы массива. Для выполнения пользовательских сравнений может быть использована дополнительная функция сравнения. Если функция сравнения не задана, выполняется сортировка по алфавиту
<code>splice(index, count)</code>	Удаляет/заменяет указанное в параметре <code>count</code> количество элементов массива, начиная с указанного в параметре <code>index</code> элемента
<code>every(test)</code>	Для всех элементов массива вызывает функцию <code>test</code> . Возвращает <code>true</code> , если функция возвращает <code>true</code> для всех элементов, в противном случае — <code>false</code>
<code>some(test)</code>	Проверяет, удовлетворяет ли хотя бы один элемент массива условию, заданному в функции <code>test</code> . Возвращает <code>true</code> , если хотя бы один элемент соответствует условию, в противном случае — <code>false</code>
<code>filter(test)</code>	Возвращает новый массив, содержащий только те элементы, для которых функция <code>test</code> возвращает <code>true</code>
<code>find(test)</code>	Возвращает первый элемент массива, для которого функция <code>test</code> возвращает <code>true</code>
<code>findIndex(test)</code>	Возвращает индекс (порядковый номер) первого элемента массива, для которого функция <code>test</code> возвращает <code>true</code>
<code>forEach(callback)</code>	Вызывает функцию обратного вызова (<code>callback</code>) для каждого элемента массива, как было описано ранее
<code>includes(value)</code>	Возвращает <code>true</code> , если массив содержит указанное значение <code>value</code>
<code>map(callback)</code>	Возвращает новый массив, содержащий результат вызова функции обратного вызова (<code>callback</code>) для каждого элемента массива
<code>reduce(callback)</code>	Возвращает аккумулированное значение, полученное в результате вызова функции обратного вызова (<code>callback</code>) для каждого элемента массива

3.4.1. Оператор расширения spread для массивов

Оператор spread позволяет расширить содержимое массива так, чтобы его элементы можно было использовать в качестве аргументов функции. Оператор spread представлен в виде троеточия (...). В листинге 3.21 мы использовали его для передачи содержимого массива в функцию sumPrices.

```
...
let totalPrice = sumPrices(...prices);
...
```

Данный оператор используется перед именем массива. Кроме того, он также может быть использован для расширения содержимого массива с целью упрощения конкатенации, как показано в листинге 3.22.

Листинг 3.22. Использование оператора spread в файле index.js

```
let names = ["Hat", "Boots", "Gloves"];
let prices = [];

prices.push(100);
prices.push("100");
prices.push(50.25);

console.log('First Item: ${names[0]}: ${prices[0]}');

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
    total + (Number.isNaN(Number(val)) ? 0 : Number(val)));

let totalPrice = sumPrices(...prices);
console.log('Total: ${totalPrice} ${typeof totalPrice');

let combinedArray = [...names, ...prices];
combinedArray.forEach(element =>
    console.log('Combined Array Element: ${element}'));
```

В листинге 3.22 оператор spread используется для создания нового массива, который содержит элементы массивов names и prices, в результате чего получается следующий вывод:

```
First Item: Hat: 100
Total: 250.25 number
Combined Array Element: Hat
Combined Array Element: Boots
Combined Array Element: Gloves
Combined Array Element: 100
Combined Array Element: 100
Combined Array Element: 50.25
```

3.4.2. Деструктуризация массивов

Деструктурирующее присваивание позволяет извлекать значения из массивов и присваивать их новым переменным, как показано в листинге 3.23.

Листинг 3.23. Деструктуризация массива в файле index.js

```
let names = ["Hat", "Boots", "Gloves"];  
  
let [one, two] = names;  
console.log('One: ${one}, Two: ${two}');
```

В левой части выражения указаны переменные, которым будут присвоены значения. В нашем примере первый элемент массива `names` будет присвоен переменной с именем `one`, второй — переменной `two`. Количество переменных не обязательно должно совпадать с количеством элементов в массиве: любые элементы, для которых нет переменных в деструктурирующем присваивании, игнорируются, а любым переменным в деструктурирующем присваивании, для которых нет соответствующего элемента массива, будет присвоено значение `undefined`. Код, приведенный в листинге 3.23, выдает следующий результат:

```
One: Hat, Two: Boots
```

Игнорирование элементов при деструктуризации массива

Переменные можно игнорировать, не указывая их имена в присваивании (листинг 3.24).

Листинг 3.24. Игнорирование элементов в файле index.js

```
let names = ["Hat", "Boots", "Gloves"];  
  
let [, , three] = names;  
console.log('Three: ${three}');
```

В первых двух позициях присваивания имя не указано, а это означает, что первые два элемента массива игнорируются. Третий элемент присваивается переменной с именем `three`, и код выдает следующий результат:

```
Three: Gloves
```

Присвоение оставшихся элементов массиву

Последнее имя переменной в деструктурирующем присваивании может иметь префикс из трех точек (...), известный как *выражение rest*, или *шаблон rest*, который присваивает все оставшиеся элементы массиву, как показано в листинге 3.25. (Для согласованности выражение `rest` часто называют *оператором spread*, поскольку оба обозначаются троеточием и ведут себя одинаково.)

Листинг 3.25. Присваивание оставшихся элементов в файле index.js

```
let names = ["Hat", "Boots", "Gloves"];  
  
let [, , three] = names;  
console.log('Three: ${three}');  
  
let prices = [100, 120, 50.25];  
let [...highest] = prices.sort((a, b) => a - b);  
highest.forEach(price => console.log('High price: ${price}'));
```

Массив `prices` сортируется, первый элемент отбрасывается, а оставшиеся элементы присваиваются массиву с именем `highest`, который нумеруется так, чтобы значения можно было выводить на консоль. Это дает следующий результат:

```
Three: Gloves
High price: 100
High price: 120
```

3.5. РАБОТА С ОБЪЕКТАМИ

Объекты JavaScript представляют собой коллекции свойств, каждое из которых имеет имя и значение. Самый простой способ определения объекта – использовать синтаксис литералов, как показано в листинге 3.26.

Листинг 3.26. Создание объекта в файле index.js

```
let hat = {
  name: "Hat",
  price: 100
};

let boots = {
  name: "Boots",
  price: "100"
}

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
  total + (Number.isNaN(Number(val)) ? 0 : Number(val)));
  
let totalPrice = sumPrices(hat.price, boots.price);
console.log('Total: ${totalPrice} ${typeof totalPrice}');
```

В синтаксисе литералов в фигурных скобках содержится список имен и значений свойств. Имена отделяются от значений двоеточиями, а от других свойств – запятыми. Объекты можно присваивать переменным, использовать в качестве аргументов функций и хранить в массивах. В листинге 3.26 определены два объекта, которые присвоены переменным `hat` и `boots`. Доступ к свойствам объекта можно получить через имя переменной, как показано в этом операторе, который получает значения свойств `price`, определенных для обоих объектов:

```
...
let totalPrice = sumPrices(hat.price, boots.price);
...
```

Код из листинга 3.26 выдает следующий результат:

```
Total: 200 number
```

3.5.1. Добавление, изменение и удаление свойств объекта

Как и все остальное в JavaScript, объекты в JavaScript являются динамическими. Свойства можно добавлять и удалять, а самим свойствам можно присваивать значения любого типа, как показано в листинге 3.27.

Листинг 3.27. Манипулирование объектом в файле index.js

```

let hat = {
  name: "Hat",
  price: 100
};

let boots = {
  name: "Boots",
  price: "100"
}

let gloves = {
  productName: "Gloves",
  price: "40"
}

gloves.name = gloves.productName;
delete gloves.productName;
gloves.price = 20;

let sumPrices = (...numbers) => numbers.reduce((total, val) =>
  total + (Number.isNaN(Number(val)) ? 0 : Number(val)));
  
let totalPrice = sumPrices(hat.price, boots.price, gloves.price);
console.log('Total: ${totalPrice} ${typeof totalPrice}');

```

Здесь мы создаем объект `gloves` со свойствами `productName` и `price`. Далее операторы задают свойство `name`, используют ключевое слово `delete` для удаления свойства и присваивают свойству `price` значение типа `number`, заменяющее предыдущее строковое значение. Код, приведенный в листинге 3.27, выдает следующий результат:

Total: 220 number

Защита от объектов и свойств типа undefined

При работе с объектами важно соблюдать осторожность, поскольку они могут иметь форму (термин, используемый для обозначения набора свойств и значений), отличную от ожидаемой или изначально использовавшейся при создании объекта.

Поскольку форма объекта может измениться, установка или чтение значений свойств, которые не были определены, не является ошибкой. Если вы попытаетесь задать несуществующее свойство, оно будет добавлено к объекту и ему будет присвоено указанное значение. Если прочитать несуществующее свойство, вернется значение `undefined`. Для того чтобы гарантировать надежную работу кода, можно воспользоваться функцией приведения типов и операторами нулевого слияния `??` или логического ИЛИ, как показано в листинге 3.28.

Листинг 3.28. Защита от значений типа `undefined` в файле index.js

```

let hat = {
  name: "Hat",
  price: 100
};

```

```

let boots = {
    name: "Boots",
    price: "100"
}

let gloves = {
    productName: "Gloves",
    price: "40"
}

gloves.name = gloves.productName;
delete gloves.productName;
gloves.price = 20;

let propertyCheck = hat.price ?? 0;
let objectAndPropertyCheck = (hat ?? {}).price ?? 0;
console.log('Checks: ${propertyCheck}, ${objectAndPropertyCheck}');

```

Вероятно, код кажется сложным, но оператор нулевого слияния ?? приводит значения `undefined` и `null` к `false`, а другие — к `true`. Эти проверки могут быть использованы для обеспечения резервного значения для отдельного свойства, объекта или их комбинации.

Первая проверка в листинге 3.28 предполагает, что переменной `hat` уже присвоено значение, но при этом проверяется, определена ли переменная `hat.price` и присвоено ли ей значение. Второй оператор более «осторожен», но его код сложнее воспринимается. Он проверяет, присвоено ли переменной `hat` значение, прежде чем проверять свойство `price`. Код в листинге 3.28 выдает следующий результат:

`Checks: 100, 100`

Вторую проверку в листинге 3.28 можно упростить, используя опциональную цепочку, как показано в листинге 3.29.

Листинг 3.29. Использование опциональной цепочки в файле index.js

```

let hat = {
    name: "Hat",
    price: 100
};

let boots = {
    name: "Boots",
    price: "100"
}

let gloves = {
    productName: "Gloves",
    price: "40"
}

gloves.name = gloves.productName;
delete gloves.productName;
gloves.price = 20;

```

```
let propertyCheck = hat.price ?? 0;
let objectAndPropertyCheck = hat?.price ?? 0;
console.log('Checks: ${propertyCheck}, ${objectAndPropertyCheck}');
```

Оператор опциональной последовательности (символ `?`) прекращает вычисление выражения, если значение, к которому он применяется, равно `null` или `undefined`. В листинге мы применили его к `hat`, поэтому выражение не будет пытаться прочитать значение свойства `price`, если `hat` не определено или равно `null`. В результате, если `hat` или `hat.price` не определены или равны `null`, будет использовано резервное значение.

3.5.2. Использование операторов spread и rest для объектов

Оператор `spread` (`...`) можно использовать для расширения свойств и значений, определяемых объектом, что позволяет легко создавать новый объект на основе свойств другого объекта, как показано в листинге 3.30.

Листинг 3.30. Использование оператора расширения объекта в файле index.js

```
let hat = {
  name: "Hat",
  price: 100
};

let boots = {
  name: "Boots",
  price: "100"
}

let otherHat = { ...hat };
console.log('Spread: ${otherHat.name}, ${otherHat.price}');
```

Оператор расширения используется для включения свойств объекта `hat` в синтаксис объектного литерала. В результате его использования образуется новый объект `otherHat`, содержащий те же свойства, что и `hat`. Код в листинге 3.30 выдает следующий результат:

```
Spread: Hat, 100
```

Оператор `spread` также допускается комбинировать с другими свойствами для добавления, замены или поглощения свойств исходного объекта, как показано в листинге 3.31.

Листинг 3.31. Добавление, замена и поглощение в файле index.js

```
let hat = {
  name: "Hat",
  price: 100
};
```

```

let boots = {
  name: "Boots",
  price: "100"
}

let additionalProperties = { ...hat, discounted: true};
console.log('Additional: ${JSON.stringify(additionalProperties)}');

let replacedProperties = { ...hat, price: 10};
console.log('Replaced: ${JSON.stringify(replacedProperties)}');

let { price , ...someProperties } = hat;
console.log('Selected: ${JSON.stringify(someProperties)}');

```

Имена и значения свойств, расширенные оператором spread, рассматриваются так же, как если бы они были указаны отдельно в синтаксисе объектного литерала. Это означает возможность изменения формы объекта путем смешивания оператора расширения с другими свойствами. Например, выражение:

```

...
let additionalProperties = { ...hat, discounted: true};
...

```

будет расширено таким образом, что свойства, определяемые объектом `hat`, объединяются со свойством `discounted`, что эквивалентно этому утверждению:

```
let additionalProperties = { name: "Hat", price: 100, discounted: true};
```

Если в синтаксисе объектного литерала имя свойства используется дважды, то будет использоваться именно второе значение. Данная возможность может быть использована для изменения значения свойства, полученного через оператор spread, и означает, что данное утверждение:

```

...
let replacedProperties = { ...hat, price: 10};
...

```

будет расширено таким образом, чтобы оно было эквивалентно выражению:

```
let replacedProperties = { name: "Hat", price: 100, price: 10};
```

В результате получится объект со свойством `name` и значением из объекта `hat`, но с новым свойством `price`, значение которого равно `10`. Оператор остаточных параметров — `rest` (обозначается теми же тремя точками, как и оператор расширения) — может применяться для выбора свойств или для их исключения при использовании синтаксиса объектного литерала. Данное утверждение определяет переменные с именами `price` и `someProperties`:

```

...
let { price , ...someProperties } = hat;
...

```

Свойства, определенные объектом `hat`, декомпозируются. Свойство `hat.price` присваивается новому свойству `price`, а все остальные — объекту `someProperties`.

Встроенный метод `JSON.stringify` создает строковое представление объекта, используя формат данных JSON. Он полезен в основном для представления простых объектов (бесполезен для работы с функциями), но он помогает понять, как составляются объекты. Код в листинге 3.31 выдает следующий результат:

```
Additional: {"name": "Hat", "price": 100, "discounted": true}
Replaced: {"name": "Hat", "price": 10}
Selected: {"name": "Hat"}
```

3.5.3. Определение геттеров и сеттеров

Геттеры (getters) и сеттеры (setters) — это функции, которые вызываются при чтении или присвоении значения свойства, как показано в листинге 3.32.

Листинг 3.32. Использование геттеров и сеттеров в файле index.js

```
let hat = {
    name: "Hat",
    _price: 100,
    priceIncTax: 100 * 1.2,

    set price(newPrice) {
        this._price = newPrice;
        this.priceIncTax = this._price * 1.2;
    },

    get price() {
        return this._price;
    }
};

let boots = {
    name: "Boots",
    price: "100",

    get priceIncTax() {
        return Number(this.price) * 1.2;
    }
}

console.log('Hat: ${hat.price}, ${hat.priceIncTax}');
hat.price = 120;
console.log('Hat: ${hat.price}, ${hat.priceIncTax}');

console.log('Boots: ${boots.price}, ${boots.priceIncTax}');
boots.price = "120";
console.log('Boots: ${boots.price}, ${boots.priceIncTax}');
```

В примере введено свойство `priceIncTax`, значение которого автоматически обновляется при установке свойства `price`. Для этого в объекте `hat` используются геттер и сеттер свойства `price` для обновления теневого свойства `_price`. Когда свойству `price` присваивается новое значение, сеттер обновляет теневое свойство и свойство `priceIncTax`. Когда считывается значение свойства `price`, геттер

возвращает значение свойства `_price`. (Теневое свойство, или *backing property*, необходимо, поскольку геттеры и сеттеры рассматриваются как свойства и не могут иметь то же имя, что и обычные свойства, определенные объектом.)

О СВОЙСТВАХ PRIVATE В JAVASCRIPT

В JavaScript нет встроенной поддержки приватных (*private*) свойств, за исключением классов (подробнее о классах — в главе 4). Под приватным подразумевается свойство, доступ к которому могут получить только методы, геттеры и сеттеры объекта. Существуют способы достижения аналогичного эффекта вне классов, однако они сложны, и поэтому наиболее часто используется соглашение об именовании для свойств, которые не предназначены для публичного использования. Это не предотвращает доступ к подобным свойствам, но по крайней мере делает очевидным то, что их применение не желательно. Широко распространенным соглашением об именовании является добавление префикса к имени свойства в виде символа подчеркивания (например, `_price` в листинге 3.32). Данный прием не требуется при разработке на языке TypeScript, который имеет собственный подход к приватным свойствам (подробнее об этом — в главе 11).

Объект `boots` определяет то же поведение, что и объект `hat`, но при этом создает геттер, не имеющий соответствующего сеттера, что позволяет считывать значение, но не изменять его. Таким образом, данный пример демонстрирует, что геттеры и сеттеры не обязательно должны использоваться вместе. Код в листинге 3.32 выдает следующий результат:

```
Hat: 100, 120
Hat: 120, 144
Boots: 100, 120
Boots: 120, 144
```

3.5.4. Определение методов

Поначалу JavaScript может показаться запутанным, но при более глубоком изучении начинает просматриваться внутренняя логика, которая не всегда очевидна при обычном использовании. Одним из таких примеров являются методы, которые основываются на возможностях, рассмотренных в предыдущих разделах, как показано в листинге 3.33.

Листинг 3.33. Определение методов в файле index.js

```
let hat = {
  name: "Hat",
  _price: 100,
  priceIncTax: 100 * 1.2,
```

```

set price(newPrice) {
    this._price = newPrice;
    this.priceIncTax = this._price * 1.2;
},
get price() {
    return this._price;
},
writeDetails: function() {
    console.log('${this.name}: ${this.price}, ${this.priceIncTax}');
}
};

let boots = {
    name: "Boots",
    price: "100",
    get priceIncTax() {
        return Number(this.price) * 1.2;
    }
}

hat.writeDetails();
hat.price = 120;
hat.writeDetails();

console.log('Boots: ${boots.price}, ${boots.priceIncTax}');
boots.price = "120";
console.log('Boots: ${boots.price}, ${boots.priceIncTax}');

```

Метод в JavaScript – это свойство, значением которого является функция, то есть для методов доступны все возможности, предоставляемые функциями, например параметры по умолчанию и пр. Метод в листинге 3.33 определен с помощью ключевого слова `function`, однако существует и более лаконичный синтаксис, как показано в листинге 3.34.

Листинг 3.34. Использование краткого синтаксиса в файле index.js

```

...
writeDetails() {
    console.log('${this.name}: ${this.price}, ${this.priceIncTax}');
}
...

```

Здесь опущены ключевое слово `function` и двоеточие, отделяющее имя свойства от его значения, что позволяет определять методы в стиле, который является естественным для многих разработчиков. Приведенные в этом разделе листинги дают следующий результат:

```

Hat: 100, 120
Hat: 120, 144
Boots: 100, 120
Boots: 120, 144

```

3.6. КЛЮЧЕВОЕ СЛОВО THIS

Ключевое слово `this` может поставить в тупик даже опытных программистов JavaScript. В других языках программирования оно используется для обозначения текущего экземпляра объекта, созданного на основе класса. В JavaScript ключевое слово `this` часто может работать похожим образом — ровно до того момента, когда какое-либо изменение не приведет к сбоям в приложении и появлению значений типа `undefined`.

Для демонстрации этого переопределим метод в объекте `hat` с помощью синтаксиса стрелочной функции, как показано в листинге 3.35.

Листинг 3.35. Использование синтаксиса стрелочной функции в файле index.js

```
let hat = {
    name: "Hat",
    _price: 100,
    priceIncTax: 100 * 1.2,

    set price(newPrice) {
        this._price = newPrice;
        this.priceIncTax = this._price * 1.2;
    },

    get price() {
        return this._price;
    },

    writeDetails: () =>
        console.log(`${this.name}: ${this.price}, ${this.priceIncTax}`)
};

let boots = {
    name: "Boots",
    price: "100",

    get priceIncTax() {
        return Number(this.price) * 1.2;
    }
}

hat.writeDetails();
hat.price = 120;
hat.writeDetails();

console.log('Boots: ${boots.price}, ${boots.priceIncTax}');
boots.price = "120";
console.log('Boots: ${boots.price}, ${boots.priceIncTax}');
```

Здесь используется тот же оператор `console.log`, что и в листинге 3.34, но после сохранения изменений код при выполнении выдаст следующий результат, содержащий неопределенные значения:

```
undefined: undefined, undefined
undefined: undefined, undefined
Boots: 100, 120
Boots: 120, 144
```

Чтобы разобраться в этой проблеме и устраниить ее, необходимо сделать шаг назад и изучить, что на самом деле делает ключевое слово `this` в JavaScript.

3.6.1. Ключевое слово `this` в автономных функциях

Ключевое слово `this` может быть использовано в любой функции, даже если она не используется в качестве метода, как показано в листинге 3.36.

Листинг 3.36. Вызов функции в файле index.js

```
function writeMessage(message) {  
    console.log(`${this.greeting}, ${message}`);  
}  
  
greeting = "Hello";  
  
writeMessage("It is sunny today");
```

Функция `writeMessage` считывает свойство с именем `greeting` из `this` в одно из выражений строки шаблона, передаваемого методу `console.log`. Хотя ключевое слово `this` больше не встречается в листинге, после сохранения и выполнения кода получается следующий результат:

```
Hello, It is sunny today
```

JavaScript определяет глобальный объект, которому можно присвоить значения, доступные во всем приложении. Глобальный объект используется для обеспечения доступа к важным функциям в среде выполнения, таким как объект `document` в браузерах, который позволяет взаимодействовать с API объектной модели документа (Document Object Model, DOM).

Значения, которым присвоены имена без использования ключевых слов `let`, `const` или `var`, присваиваются глобальному объекту. Оператор, присваивающий строковое значение `Hello`, создает переменную в глобальной области видимости. При выполнении функции ключевое слово `this` получает значение глобального объекта, поэтому чтение `this.greeting` возвращает строковое значение `Hello`, что и объясняет вывод, сгенерированный приложением.

Стандартным способом вызова функции является использование круглых скобок, содержащих аргументы, но в JavaScript данный синтаксис преобразуется в оператор, показанный в листинге 3.37.

Листинг 3.37. Вызов функции в файле index.js

```
function writeMessage(message) {  
    console.log(`${this.greeting}, ${message}`);  
}  
greeting = "Hello";  
  
writeMessage("It is sunny today");  
writeMessage.call(global, "It is sunny today");
```

Как уже говорилось, функции в JavaScript являются объектами и имеют свои методы, включая метод `call`, который позволяет осуществить принудительный вызов функции. Первым аргументом метода `call` является значение `this`, которое

устанавливается в глобальный объект. Вот почему ключевое слово `this` может использоваться в любой функции, и именно по этой причине по умолчанию оно возвращает глобальный объект.

Новый оператор в листинге 3.37 использует метод `call` напрямую и устанавливает значение `this` глобальному объекту с тем же результатом, что и обычный вызов функции перед ним, что видно из следующего вывода, выдаваемого кодом при выполнении:

```
Hello, It is sunny today
Hello, It is sunny today
```

Имя глобального объекта меняется в зависимости от среды выполнения. В коде, выполняемом Node.js, используется имя `global`, но в браузерах может потребоваться `window` или `self`. На момент написания книги в рядах сообщества звучала идея по стандартизации имени `global`, однако она еще не получила всеобщего признания.

ОБ ЭФФЕКТЕ СТРОГОГО РЕЖИМА

JavaScript поддерживает строгий режим, который отключает или ограничивает возможности, исторически ставшие причиной появления некачественного программного обеспечения или мешавшие среде выполнения эффективно выполнять код. Когда включен строгий режим, значение по умолчанию для `this` не определено (`undefined`), что предотвращает случайное использование глобального объекта, а значения с глобальной областью видимости должны быть явно определены как свойства глобального объекта. Дополнительную информацию см. на сайте https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode. Компилятор TypeScript предоставляет возможность автоматического включения строгого режима в генерируемый им JavaScript-код (подробнее об этом в главе 5).

3.6.2. Ключевое слово `this` в методах

Когда функция вызывается как метод объекта, значение `this` устанавливается в объект, как показано в листинге 3.38.

Листинг 3.38. Вызов функции в качестве метода в файле index.js

```
let myObject = {
  greeting: "Hi, there",
  writeMessage(message) {
    console.log(`${this.greeting}, ${message}`);
  }
}
greeting = "Hello";
myObject.writeMessage("It is sunny today");
```

Когда функция вызывается через объект, оператор, вызывающий ее, эквивалентен использованию метода `call` с объектом в качестве первого аргумента, например, так:

```
...
myObject.writeMessage.call(myObject, "It is sunny today");
...
```

Будьте осторожны, поскольку параметр `this` устанавливается иначе, если к функции обращаются вне ее объекта. Это может произойти, если функцию присваивают переменной, как показано в листинге 3.39.

Листинг 3.39. Вызов функции в файле index.js

```
let myObject = {
    greeting: "Hi, there",
    writeMessage(message) {
        console.log(`${this.greeting}, ${message}`);
    }
}

greeting = "Hello";

myObject.writeMessage("It is sunny today");

let myFunction = myObject.writeMessage;
myFunction("It is sunny today");
```

Функции можно использовать как любые другие значения, в том числе присваивать их переменным вне объекта, в котором они были определены, как показано в листинге. Если функцию вызывают через переменную, то `this` устанавливается в глобальный объект. Это часто приводит к проблемам, когда функции используются в качестве аргументов других методов или обратных вызовов для обработки событий, в результате чего одна и та же функция ведет себя по-разному в зависимости от способа ее вызова, как показано в результате выполнения кода из листинга 3.39:

```
Hi, there, It is sunny today
Hello, It is sunny today
```

3.6.3. Изменение поведения ключевого слова `this`

Одним из способов управления значением `this` является вызов функций с помощью метода `call`, но это не очень удобно и требует дополнительных действий при каждом вызове функции. Более надежный способ — использовать метод `bind`. Он позволяет установить значение `this` независимо от способа вызова функции (листинг 3.40).

Листинг 3.40. Установка значения `this` в файле index.js

```
let myObject = {
    greeting: "Hi, there",
```

```

        writeMessage(message) {
            console.log(`${this.greeting}, ${message}`);
        }
    }

myObject.writeMessage = myObject.writeMessage.bind(myObject);

greeting = "Hello";

myObject.writeMessage("It is sunny today");

let myFunction = myObject.writeMessage;
myFunction("It is sunny today");

```

Метод `bind` возвращает новую функцию, которая при вызове будет иметь постоянное значение `this`. Функция, возвращаемая при помощи метода `bind`, используется для замены исходного метода, обеспечивая согласованность при вызове метода `writeMessage`. Однако использование `bind` неудобно тем, что ссылка на объект доступна только после его создания, что приводит к двухэтапному процессу создания объекта и последующего вызова `bind` для замены каждого из методов, для которых требуется постоянное значение `this`. Код в листинге 3.40 выдает следующий результат:

```

Hi, there, It is sunny today
Hi, there, It is sunny today

```

Значением `this` всегда является `myObject`, даже если `writeMessage` вызывается как отдельная функция.

3.6.4. Ключевое слово `this` в стрелочных функциях

Еще одна сложность использования `this` заключается в том, что стрелочные функции работают не так, как обычные функции. Они не имеют собственного значения `this` и наследуют ближайшее значение `this`, которое они могут найти при выполнении. Чтобы продемонстрировать, как это работает, в пример из листинга 3.41 добавлена стрелочная функция.

Листинг 3.41. Использование стрелочной функции в файле index.js

```

let myObject = {
    greeting: "Hi, there",

    getWriter() {
        return (message) => console.log(`${this.greeting}, ${message}`);
    }
}

greeting = "Hello";

let writer = myObject.getWriter();
writer("It is raining today");

let standAlone = myObject.getWriter;
let standAloneWriter = standAlone();
standAloneWriter("It is sunny today");

```

В листинге 3.41 `getWriter` представляет собой обычную функцию, которая возвращает в качестве результата стрелочную функцию. Когда вызывается стрелочная функция, возвращаемая `getWriter`, она продвигается вверх по области видимости, пока не найдет значение `this`. Как следствие, способ вызова функции `getWriter` определяет значение `this` для стрелочной функции. Вот первые два оператора, вызывающие функции:

```
...
let writer = myObject.getWriter();
writer("It is raining today");
...
```

Их можно объединить следующим образом:

```
...
myObject.getWriter()["It is raining today"];
...
```

Комбинированное выражение выше может показаться немного сложным, но оно подчеркивает, что значение `this` зависит от того, как вызывается функция. Метод `getWriter` вызывается через `myObject`, а это говорит о том, что значение `this` будет установлено в `myObject`. Когда вызывается стрелочная функция, она находит значение `this` из функции `getWriter`. В результате, когда метод `getWriter` вызывается через `myObject`, значение `this` в стрелочной функции будет равно `myObject`, а выражение `this.greeting` в строке шаблона будет иметь значение `Hi, there`.

Операторы второго набора рассматривают `getWriter` как отдельную функцию, поэтому `this` будет установлено для глобального объекта. При вызове стрелочной функции выражение `this.greeting` будет равно `Hello`. Код в листинге 3.41 выдает следующий результат, подтверждая значение `this` в каждом случае:

```
Hi, there, It is raining today
Hello, It is sunny today
```

3.6.5. Возвращаясь к исходной проблеме

Данный раздел начался с переопределения функции в синтаксисе стрелки и демонстрации того, что она ведет себя по-другому, выдавая на выходе значение `undefined`. Вот объект и его функция:

```
...
let hat = {
  name: "Hat",
  _price: 100,
  priceIncTax: 100 * 1.2,

  set price(newPrice) {
    this._price = newPrice;
    this.priceIncTax = this._price * 1.2;
  },

  get price() {
    return this._price;
  },
}
```

```

    writeDetails: () =>
      console.log(`${this.name}: ${this.price}, ${this.priceIncTax}`)
};

...

```

Поведение кода изменилось потому, что стрелочные функции не имеют собственного значения `this` и они не находятся внутри обычной функции, которая могла бы предоставить это значение. Чтобы решить эту проблему и обеспечить стабильные результаты, нам следует вернуться к обычной функции и использовать метод `bind`, чтобы исправить значения `this`, как показано в листинге 3.42.

Листинг 3.42. Решение проблемы с функциями в файле index.js

```

let hat = {
  name: "Hat",
  _price: 100,
  priceIncTax: 100 * 1.2,

  set price(newPrice) {
    this._price = newPrice;
    this.priceIncTax = this._price * 1.2;
  },

  get price() {
    return this._price;
  },

  writeDetails() {
    console.log(`${this.name}: ${this.price}, ${this.priceIncTax}`);
  }
};

let boots = {
  name: "Boots",
  price: "100",

  get priceIncTax() {
    return Number(this.price) * 1.2;
  }
}

hat.writeDetails = hat.writeDetails.bind(hat);
hat.writeDetails();
hat.price = 120;
hat.writeDetails();

console.log('Boots: ${boots.price}, ${boots.priceIncTax}');
boots.price = "120";
console.log('Boots: ${boots.price}, ${boots.priceIncTax}');

```

С учетом этих изменений значением `this` для метода `writeDetails` будет являться содержащий его объект, независимо от способа его вызова, что приведет к следующему результату:

```

Hat: 100, 120
Hat: 120, 144
Boots: 100, 120
Boots: 120, 144

```

РЕЗЮМЕ

В этой главе вы познакомились с основными возможностями системы типов JavaScript. Они часто сбивают с толку, поскольку работают немного иначе, чем в других языках программирования. Понимание этих возможностей упрощает работу с TypeScript, помогая понять, какие проблемы решает TypeScript. В JavaScript существует набор встроенных типов данных, которые используются для представления различных значений.

- JavaScript пытается преобразовывать типы данных, когда они объединяются с оператором.
- Функции в JavaScript могут быть определены с помощью литерального синтаксиса, в котором объявляются параметры и тело функции, а также с использованием синтаксиса толстой стрелки или лямбда-функции.
- Функции в JavaScript могут принимать переменное количество аргументов, которое можно определить с помощью параметра rest.
- Функции в JavaScript формально не объявляют возвращаемый результат и могут возвращать результаты любого типа.
- Массивы в JavaScript имеют переменную длину и могут принимать значения любого типа.
- Объекты в JavaScript представляют собой набор свойств и значений и определяются с помощью литерального синтаксиса.
- Объекты в JavaScript можно изменять путем добавления, изменения или удаления свойств.
- Объекты в JavaScript могут быть определены с помощью методов, которые представляют собой функции, назначаемые свойству.
- Ключевое слово `this` относится к различным объектам в зависимости от способа вызова функции.

В следующей главе мы рассмотрим дополнительные возможности типов JavaScript, необходимые для понимания TypeScript.

Обзор JavaScript, часть 2

В этой главе

- ✓ Работа с прототипами объектов в JavaScript.
- ✓ Определение классов в JavaScript.
- ✓ Генерация и использование последовательностей.
- ✓ Использование коллекций в JavaScript.
- ✓ Создание и использование модулей в JavaScript.

В данной главе мы продолжим рассматривать возможности JavaScript, важные для разработки на TypeScript. Основное внимание здесь уделяется поддержке объектов в JavaScript, различным способам их определения и их связи с классами JavaScript. Также будут продемонстрированы подходы к работе с последовательностями значений, коллекциями JavaScript и функцией модулей, позволяющей разделить проект на несколько JavaScript-файлов.

4.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В этой главе мы все еще будем использовать проект `primer`, созданный в главе 3. Прежде чем приступить к работе, замените содержимое файла `index.js` в папке `primer` кодом из листинга 4.1.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 4.1. Замена кода в файле index.js из папки primer

```
let hat = {
  name: "Hat",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

console.log('Hat: ${hat.price}, ${hat.getPriceIncTax() }');
```

Откройте новое окно командной строки, перейдите в папку primer и выполните команду из листинга 4.2, чтобы начать мониторинг и выполнение JavaScript-файла.

Листинг 4.2. Запуск средств разработки

```
npx nodemon index.js
```

Пакет nodemon выполнит содержимое файла index.js и выдаст следующий результат:

```
[nodemon] 2.0.20
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching: ***!
[nodemon] starting 'node index.js'
Hat: 100, 120
[nodemon] clean exit - waiting for changes before restart
```

4.2. НАСЛЕДОВАНИЕ ОБЪЕКТОВ В JAVASCRIPT

Объекты JavaScript связаны с другими объектами, известными как *прототипы*, от которых они наследуют свойства и методы. Поскольку прототипы сами являются объектами и могут иметь свои прототипы, объекты формируют цепочку наследования, что позволяет один раз определить сложные функции и последовательно использовать их.

Когда объект создается с помощью синтаксиса литерала, как в примере с объектом hat в листинге 4.1, его прототипом является Object — встроенный в JavaScript объект. Object предоставляет основные функции, которые наследуются всеми объектами, включая метод `toString`, который возвращает строковое представление объекта, как показано в листинге 4.3.

Листинг 4.3. Использование объекта в файле index.js из папки primer

```
let hat = {
  name: "Hat",
  price: 100,
  getPriceIncTax() {
    return Number(this.price) * 1.2;
  }
};

console.log('Hat: ${hat.price}, ${hat.getPriceIncTax() }');
console.log('toString: ${hat.toString()}');
```

Первый оператор `console.log` принимает строку шаблона, включающую свойство `price`, которое является одним из свойств объекта `hat`. Новый оператор вызывает метод `toString`. Ни одно из свойств объекта `hat` не имеет имени `toString`, поэтому среда выполнения JavaScript обращается к прототипу объекта `hat`, которым является `Object` и который предоставляет свойство `toString`, что приводит к следующему результату:

```
Hat: 100, 120
toString: [object Object]
```

Результат, возвращенный методом `toString`, не особенно полезен, но он иллюстрирует связь между объектом `hat` и его прототипом, как показано на рис. 4.1.



Рис. 4.1. Объект и его прототип

4.2.1. Проверка и изменение прототипа объекта

`Object` служит прототипом для большинства объектов, но он также предоставляет методы, которые используются напрямую, а не через наследование, и могут быть использованы для получения информации о прототипах. В табл. 4.1 описаны наиболее полезные из этих методов.

Таблица 4.1. Полезные методы объектов

Название	Описание
<code>getPrototypeOf</code>	Возвращает прототип объекта
<code>setPrototypeOf</code>	Изменяет прототип объекта
<code>getOwnPropertyNames</code>	Возвращает имена свойств объекта

В листинге 4.4 метод `getPrototypeOf` используется для подтверждения того, что два объекта, созданные с помощью синтаксиса литералов, имеют один и тот же прототип.

Листинг 4.4. Сравнение прототипов в файле index.js

```

let hat = {
    name: "Hat",
    price: 100,
    getPriceInTax() {
        return Number(this.price) * 1.2;
    }
};

let boots = {
    name: "Boots",
  
```

```

price: 100,
getPriceIncTax() {
    return Number(this.price) * 1.2;
}
}

let hatPrototype = Object.getPrototypeOf(hat);
console.log('Hat Prototype: ${hatPrototype}');

let bootsPrototype = Object.getPrototypeOf(boots);
console.log('Boots Prototype: ${bootsPrototype}');

console.log('Common prototype: ${ hatPrototype === bootsPrototype }');

console.log('Hat: ${hat.price}, ${hat.getPriceIncTax() }');
console.log('toString: ${hat.toString()}');

```

В листинге вводится другой объект и сравнивается его прототип, что дает следующий результат:

```

Hat Prototype: [object Object]
Boots Prototype: [object Object]
Common prototype: true
Hat: 100, 120
toString: [object Object]

```

Результат показывает, что объекты `hat` и `boots` имеют один и тот же прототип, как показано на рис. 4.2.

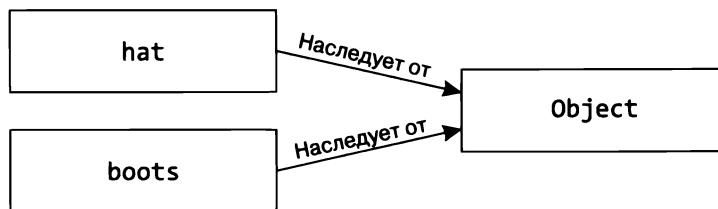


Рис. 4.2. Объекты и общий прототип

Поскольку прототипы в JavaScript – это обычные объекты, в прототипах можно определять новые свойства, а уже существующим свойствам присваивать новые значения, как показано в листинге 4.5.

Листинг 4.5. Изменение свойства прототипа в файле index.js

```

let hat = {
    name: "Hat",
    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
};

let boots = {
    name: "Boots",

```

```

    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
}

let hatPrototype = Object.getPrototypeOf(hat);
hatPrototype.toString = function() {
    return 'toString: Name: ${this.name}, Price: ${this.price}';
}

console.log(hat.toString());
console.log(boots.toString());

```

В листинге 4.5 методу `toString` назначена новая функция через прототип объекта `hat`. Поскольку объекты сохраняют ссылку на свой прототип, новый метод `toString` будет использоваться и для объекта `boots`, о чем свидетельствует следующий вывод:

```

toString: Name: Hat, Price: 100
toString: Name: Boots, Price: 100

```

4.2.2. Пользовательские прототипы

Изменения в `Object` следует вносить осторожно, поскольку они влияют на все остальные объекты приложения. Новая функция `toString` из листинга 4.5 выводит более полезный результат для объектов `hat` и `boots`, но предполагает наличие свойств `name` и `price`, которых не будет при вызове `toString` для других объектов.

Более эффективным подходом является создание пользовательского прототипа специально для объектов, про которые известно, что они имеют свойства `name` и `price`. Это можно сделать с помощью метода `Object.setPrototypeOf`, как показано в листинге 4.6.

Листинг 4.6. Использование пользовательского прототипа в файле index.js

```

let ProductProto = {
    toString: function() {
        return 'toString: Name: ${this.name}, Price: ${this.price}';
    }
}

let hat = {
    name: "Hat",
    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
};

let boots = {
    name: "Boots",
    price: 100,
    getPriceIncTax() {

```

```

        return Number(this.price) * 1.2;
    }

Object.setPrototypeOf(hat, ProductProto);
Object.setPrototypeOf(boots, ProductProto);

console.log(hat.toString());
console.log(boots.toString());

```

Прототипы могут быть определены так же, как и любой другой объект. В примере выше в качестве прототипа для объектов `hat` и `boots` используется объект `ProductProto`, определяющий метод `toString`. Объект `ProductProto` не отличается от других объектов, и у него тоже есть прототип, которым является `Object`, как показано на рис. 4.3.

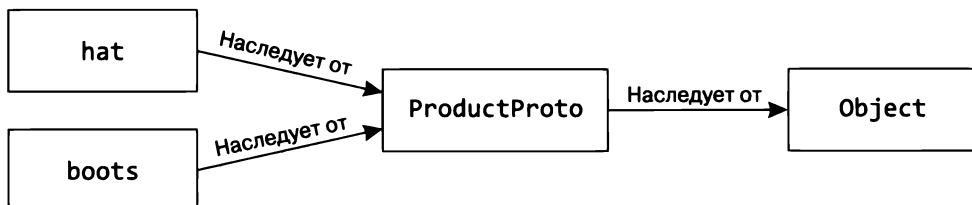


Рис. 4.3. Цепочка прототипов

Таким образом, создается цепочка прототипов, по которой JavaScript работает до тех пор, пока не найдет нужный элемент или не достигнет конца цепочки. Код в листинге 4.6 выдает следующий результат:

```

toString: Name: Hat, Price: 100
toString: Name: Boots, Price: 100

```

4.2.3. Функции-конструкторы

Функция-конструктор используется для создания нового объекта, настройки его свойств и присвоения прототипа, причем все это делается за один шаг с помощью ключевого слова `new`. Функции-конструкторы можно использовать для обеспечения единообразного создания объектов и применения правильного прототипа, как показано в листинге 4.7.

Листинг 4.7. Использование функции-конструктора в файле index.js

```

let Product = function(name, price) {
    this.name = name;
    this.price = price;
}

Product.prototype.toString = function() {
    return `toString: Name: ${this.name}, Price: ${this.price}`;
}

```

```
let hat = new Product("Hat", 100);
let boots = new Product("Boots", 100);

console.log(hat.toString());
console.log(boots.toString());
```

Функции-конструкторы вызываются с помощью ключевого слова `new`, за которым следует имя функции или ее переменной и аргументы, которые будут использоваться для конфигурирования объекта, например, так:

```
...
let hat = new Product("Hat", 100);
...
```

Среда выполнения JavaScript создает новый объект и использует его в качестве значения `this` для вызова функции-конструктора, предоставляя значения аргументов в качестве параметров. Функция-конструктор может настраивать свойства объекта с помощью ключевого слова `this`, которое установлено для нового объекта.

```
...
let Product = function(name, price) {
    this.name = name;
    this.price = price;
}
...
```

Прототипом нового объекта устанавливается объект, возвращаемый свойством `prototype` функции-конструктора. Это приводит к тому, что конструкторы определяются в двух частях: сама функция применяется для настройки свойств объекта, а объект, возвращаемый свойством `prototype`, используется для свойств и методов, которые должны быть общими для всех объектов, создаваемых конструктором. В листинге к прототипу функции-конструктора `Product` добавлено свойство `toString` для определения метода:

```
...
Product.prototype.toString = function() {
    return `ToString: Name: ${this.name}, Price: ${this.price}`;
}
...
```

Результат тот же, что и в предыдущем примере, но благодаря функции-конструктору обеспечивается последовательное создание объектов и правильное задание их прототипов.

4.2.4. Цепочка функций-конструкторов

Использование метода `setPrototypeOf` для создания цепочки пользовательских прототипов не представляет сложности. Однако для достижения того же эффекта с помощью функций-конструкторов требуется немного больше усилий, чтобы гарантировать, что объекты правильно настраиваются функциями

и получают правильные прототипы в цепочке. В листинге 4.8 представлена новая функция-конструктор, которая используется для создания цепочки с конструктором `Product`.

Листинг 4.8. Цепочка функций-конструкторов в файле index.js

```
let Product = function(name, price) {
    this.name = name;
    this.price = price;
}

Product.prototype.toString = function() {
    return 'toString: Name: ${this.name}, Price: ${this.price}';
}

let TaxedProduct = function(name, price, taxRate) {
    Product.call(this, name, price);
    this.taxRate = taxRate;
}
Object.setPrototypeOf(TaxedProduct.prototype, Product.prototype);

TaxedProduct.prototype.getPriceIncTax = function() {
    return Number(this.price) * this.taxRate;
}

TaxedProduct.prototype.toTaxString = function() {
    return '${this.toString()}, Tax: ${this.getPriceIncTax()}';
}

let hat = new TaxedProduct("Hat", 100, 1.2);
let boots = new Product("Boots", 100);

console.log(hat.toTaxString());
console.log(boots.toString());
```

Чтобы выстроить конструкторы и их прототипы в цепочку, необходимо выполнить два шага. Первый — использовать метод `call` для вызова следующего конструктора, чтобы новые объекты создавались правильно. Если мы хотим, чтобы конструктор `TaxedProduct` основывался на конструкторе `Product`, нужно использовать `call` для функции `Product` — она будет добавлять его свойства к новым объектам:

```
...
Product.call(this, name, price);
...
```

Метод `call` позволяет передавать новый объект в следующий конструктор через значение `this`.

Второй шаг — связать прототипы между собой:

```
...
Object.setPrototypeOf(TaxedProduct.prototype, Product.prototype);
...
```

Обратите внимание, что аргументами метода `setPrototypeOf` являются объекты, возвращаемые свойствами прототипа функций-конструкторов, а не сами функции. Связывание прототипов гарантирует, что при поиске свойств, не принадлежащих объекту, среда выполнения JavaScript будет следовать по цепочке. На рис. 4.4 показан новый набор прототипов.

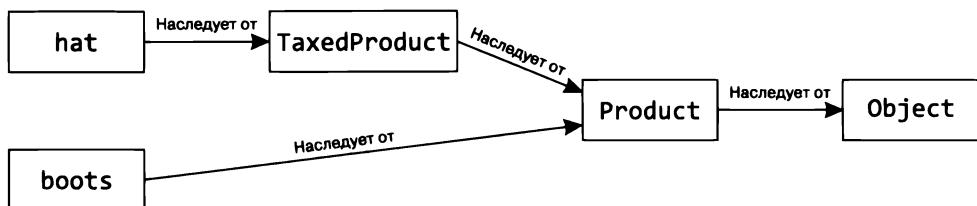


Рис. 4.4. Более сложная цепочка прототипов

Прототип `TaxedProduct` определяет метод `toTaxString`, вызывающий `toString`, который будет найден средой выполнения JavaScript в прототипе `Product`, и код в листинге 4.8 выдаст следующий результат:

```
toString: Name: Hat, Price: 100, Tax: 120
toString: Name: Boots, Price: 100
```

ДОСТУП К ПЕРЕОПРЕДЕЛЕННЫМ МЕТОДАМ ПРОТОТИПА

Прототип может переопределить свойство или метод, используя то же имя, что и имя, определенное далее по цепочке. Эта методика также известна как затенение в JavaScript и использует преимущества способа, которым среда выполнения JavaScript следует за цепочкой.

Требуется осторожность при создании переопределенного метода, доступ к которому должен осуществляться через определяющий его прототип. Прототип `TaxedProduct` может определять метод `toString`, который переопределяет метод, заданный прототипом `Product`, и может вызывать переопределенный метод, обращаясь к методу непосредственно через прототип и используя `call` для установки значения `this`.

```
...
TaxedProduct.prototype.toString = function() {
    let chainResult = Product.prototype.toString.call(this);
    return `${chainResult}, Tax: ${this.getPriceIncTax()}`;
}
...
```

Этот метод получает результат от метода `Product` прототипа `toString` и объединяет его с дополнительными данными в строке шаблона.

4.2.5. Проверка типов прототипов

Оператор `instanceof` используется для проверки того, входит ли прототип конструктора в цепочку прототипов конкретного объекта (листинг 4.9).

Листинг 4.9. Проверка прототипов в файле index.js

```
let Product = function(name, price) {
    this.name = name;
    this.price = price;
}

Product.prototype.toString = function() {
    return 'toString: Name: ${this.name}, Price: ${this.price}';
}

let TaxedProduct = function(name, price, taxRate) {
    Product.call(this, name, price);
    this.taxRate = taxRate;
}
Object.setPrototypeOf(TaxedProduct.prototype, Product.prototype);

TaxedProduct.prototype.getPriceIncTax = function() {
    return Number(this.price) * this.taxRate;
}

TaxedProduct.prototype.toTaxString = function() {
    return '${this.toString()}, Tax: ${this.getPriceIncTax()}';
}

let hat = new TaxedProduct("Hat", 100, 1.2);
let boots = new Product("Boots", 100);

console.log(hat.toTaxString());
console.log(boots.toString());
console.log('hat and TaxedProduct: ${ hat instanceof TaxedProduct}');
console.log('hat and Product: ${ hat instanceof Product}');
console.log('boots and TaxedProduct: ${ boots instanceof TaxedProduct}');
console.log('boots and Product: ${ boots instanceof Product}'');
```

Новые операторы используют `instanceof`, чтобы определить, находятся ли прототипы функций-конструкторов `TaxedProduct` и `Product` в цепочках объектов `hat` и `boots`. Код, приведенный в листинге 4.9, выдает следующий результат:

```
toString: Name: Hat, Price: 100, Tax: 120
toString: Name: Boots, Price: 100
hat and TaxedProduct: true
hat and Product: true
boots and TaxedProduct: false
boots and Product: true
```

СОВЕТ

Обратите внимание, что оператор `instanceof` используется с функцией-конструктором. Метод `Object.isPrototypeOf` применяется непосредственно с прототипами, что может быть полезно, если вы не используете конструкторы.

4.2.6. Определение статических свойств и методов

Свойства и методы, определяемые в функции-конструкторе, часто называют *статическими*, то есть доступ к ним осуществляется через конструктор, а не через отдельные объекты, созданные этим конструктором (в отличие от *свойств экземпляра*, доступ к которым осуществляется через объект). Примерами статических методов являются методы `Object.setPrototypeOf` и `Object.getPrototypeOf`. В листинге 4.10 приведен упрощенный пример, где используется статический метод.

Листинг 4.10. Определение статического метода в файле index.js

```
let Product = function(name, price) {
    this.name = name;
    this.price = price;
}

Product.prototype.toString = function() {
    return 'toString: Name: ${this.name}, Price: ${this.price}';
}

Product.process = (...products) =>
    products.forEach(p => console.log(p.toString()));

Product.process(new Product("Hat", 100, 1.2), new Product("Boots", 100));
```

Статический метод `process` определяется путем добавления нового свойства к объекту функции `Product` и присвоения ему функции. Важно помнить, что функции JavaScript являются объектами, а свойства можно свободно добавлять и удалять из объектов. Метод `process` определяет параметр `rest` и использует метод `forEach` для вызова метода `toString` для каждого полученного объекта, после чего результат выводится на консоль:

```
toString: Name: Hat, Price: 100
toString: Name: Boots, Price: 100
```

4.2.7. Классы в JavaScript

Классы в JavaScript были введены для облегчения адаптации разработчиков, работавших на других популярных языках программирования. В основе классов JavaScript лежат прототипы, поэтому они отличаются от классов в C# и Java. В листинге 4.11 удалены конструкторы и прототипы и введен класс `Product`.

Листинг 4.11. Определение класса в файле index.js

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }
}
```

```
        toString() {
            return 'toString: Name: ${this.name}, Price: ${this.price}';
        }
    }

let hat = new Product("Hat", 100);
let boots = new Product("Boots", 100);

console.log(hat.toString());
console.log(boots.toString());
```

Классы объявляются с помощью ключевого слова `class`, за которым следует имя класса. Синтаксис классов может показаться более привычным, но они транслируются в базовую систему прототипов JavaScript, описанную в предыдущем разделе.

Объекты создаются из классов с использованием ключевого слова `new`. При этом среда выполнения JavaScript создает новый объект и вызывает функцию-конструктор класса, которая получает новый объект через значение `this` и отвечает за определение свойств объекта. Методы, определяемые классами, добавляются к прототипу, назначенному объектам, которые были созданы с помощью данного класса. Код в листинге 4.11 выдает следующий результат:

```
toString: Name: Hat, Price: 100
toString: Name: Boots, Price: 100
```

Наследование в классах

Классы могут наследовать функции с помощью ключевого слова `extends` и вызывать конструктор и методы родительского класса (суперкласса) с помощью ключевого слова `super`, как показано в листинге 4.12.

Листинг 4.12. Наследование класса в файле index.js

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }
    toString() {
        return 'toString: Name: ${this.name}, Price: ${this.price}';
    }
}

class TaxedProduct extends Product {

    constructor(name, price, taxRate = 1.2) {
        super(name, price);
        this.taxRate = taxRate;
    }

    getPriceIncTax() {
        return Number(this.price) * this.taxRate;
    }
}
```

```

        toString() {
            let chainResult = super.toString();
            return `${chainResult}, Tax: ${this.getPriceIncTax()}`;
        }
    }

let hat = new TaxedProduct("Hat", 100);
let boots = new TaxedProduct("Boots", 100, 1.3);

console.log(hat.toString());
console.log(boots.toString());

```

Класс объявляет свой родительский класс с помощью ключевого слова `extends`. В примере выше класс `TaxedProduct` использует ключевое слово `extend` для наследования от класса `Product`. Ключевое слово `super` применяется в конструкторе для вызова конструктора суперкласса, что эквивалентно объединению в цепочку функций-конструкторов.

```

...
constructor(name, price, taxRate = 1.2) {
    super(name, price);
    this.taxRate = taxRate;
}
...

```

Ключевое слово `super` должно быть использовано перед ключевым словом `this`, и, как правило, оно располагается в первом операторе конструктора. Для доступа к свойствам и методам родительского класса также можно прибегнуть к ключевому слову `super`, например, так:

```

...
toString() {
    let chainResult = super.toString();
    return `${chainResult}, Tax: ${this.getPriceIncTax()}`;
}
...

```

Метод `toString`, определенный классом `TaxedProduct`, вызывает метод `toString` родительского класса, что эквивалентно переопределению методов прототипа. Код в листинге 4.12 выдает следующий результат:

```

toString: Name: Hat, Price: 100, Tax: 120
toString: Name: Boots, Price: 100, Tax: 130

```

Определение статических методов

Ключевое слово `static` используется для создания статических методов, доступ к которым осуществляется через сам класс, а не через создаваемый им объект (листинг 4.13).

Листинг 4.13. Определение статического метода в файле index.js

```

class Product {
    constructor(name, price) {
        this.name = name;
    }
}
```

```

        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

class TaxedProduct extends Product {

    constructor(name, price, taxRate = 1.2) {
        super(name, price);
        this.taxRate = taxRate;
    }

    getPriceIncTax() {
        return Number(this.price) * this.taxRate;
    }

    toString() {
        let chainResult = super.toString();
        return `${chainResult}, Tax: ${this.getPriceIncTax()}`;
    }

    static process(...products) {
        products.forEach(p => console.log(p.toString()));
    }
}

TaxedProduct.process(new TaxedProduct("Hat", 100, 1.2),
    new TaxedProduct("Boots", 100));

```

Ключевое слово `static` используется в методе `process`, который определен классом `TaxedProduct`, и доступ к нему осуществляется как `TaxedProduct.process`. Код в листинге 4.13 выдает следующий результат:

```

toString: Name: Hat, Price: 100, Tax: 120
toString: Name: Boots, Price: 100, Tax: 120

```

Создание приватных полей, свойств и методов

В последних версиях JavaScript появилась поддержка приватных членов в классах, что не позволяет использовать их за пределами класса, в котором они определены. Листинг 4.14 демонстрирует использование приватного метода.

Листинг 4.14. Приватный метод в файле index.js

```

class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

```

```

class TaxedProduct extends Product {

    constructor(name, price, taxRate = 1.2) {
        super(name, price);
        this.taxRate = taxRate;
    }

    getPriceIncTax() {
        return Number(this.price) * this.taxRate;
    }

    toString() {
        let chainResult = super.toString();
        return `${chainResult}, ${this.#getDetail()}`;
    }

    #getDetail() {
        return 'Tax: ${this.getPriceIncTax()}';
    }
}

let hat = new TaxedProduct("Hat", 100);
let boots = new TaxedProduct("Boots", 100, 1.3);

console.log(hat.toString());

console.log(boots.toString());

```

Символ `#` ставится перед именем метода для создания *хеш-имени*, которое указывает на то, что доступ к элементу класса возможен только внутри класса. Символ `#` нужен, когда используется приватный член класса, например, так:

```

...
return `${chainResult}, ${this.#getDetail()}`;
...

```

Листинг выдает тот же результат, что и предыдущий пример. Доступ к методу `#getTaxString` возможен только из класса `TaxedProduct`, а его использование в других местах, как показано в листинге 4.15, приведет к ошибке.

Листинг 4.15. Использование приватного метода в файле index.js из папки primer

```

...
console.log(hat.toString());
console.log(boots.toString());

console.log(boots.#getDetail());
...

```

При выполнении данного примера возникнет следующая ошибка:

```

...
SyntaxError: Private field '#getDetail' must be declared in an
enclosing class
...

```

4.3. ИТЕРАТОРЫ И ГЕНЕРАТОРЫ

Итераторы — это объекты, возвращающие последовательность значений. Итераторы часто используются в сочетании с коллекциями (о которых будет рассказано далее в этой главе), но они полезны и сами по себе. Итератор определяет функцию `next`, которая возвращает объект со свойствами `value` и `done`: `value` возвращает следующее значение в последовательности, а `done` устанавливается равным `true`, когда последовательность завершена. В листинге 4.16 приведен пример определения и использования итератора.

Листинг 4.16. Использование итератора в файле index.js

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

function createProductIterator() {
    const hat = new Product("Hat", 100);
    const boots = new Product("Boots", 100);
    const umbrella = new Product("Umbrella", 23);

    let lastVal;

    return {
        next() {
            switch (lastVal) {
                case undefined:
                    lastVal = hat;
                    return { value: hat, done: false };
                case hat:
                    lastVal = boots;
                    return { value: boots, done: false };
                case boots:
                    lastVal = umbrella;
                    return { value: umbrella, done: false };
                case umbrella:
                    return { value: undefined, done: true };
            }
        }
    }
}

let iterator = createProductIterator();
let result = iterator.next();
while (!result.done) {
    console.log(result.value.toString());
    result = iterator.next();
}
```

Функция `createProductIterator` возвращает объект, определяющий функцию `next`. При каждом вызове метода `next` возвращается другой объект `Product`, а затем, когда набор объектов исчерпан, возвращается объект, свойство `done` которого имеет значение `true`, что свидетельствует о достижении конца данных. Для обработки данных итератора используется цикл `while`, вызывающий метод `next` после обработки каждого объекта. Код в листинге 4.16 выдает следующий результат:

```
toString: Name: Hat, Price: 100
toString: Name: Boots, Price: 100
toString: Name: Umbrella, Price: 23
```

4.3.1. Использование генератора

Написание итераторов может быть неудобным, поскольку при каждом вызове следующей функции код должен хранить данные о состоянии, чтобы отслеживать текущую позицию в последовательности. Генераторы упрощают эту задачу, представляя собой функцию, которая вызывается только один раз и использует ключевое слово `yield` для получения значений в последовательности, как показано в листинге 4.17.

Листинг 4.17. Использование генератора в файле index.js

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

function* createProductIterator() {
    yield new Product("Hat", 100);
    yield new Product("Boots", 100);
    yield new Product("Umbrella", 23);
}

let iterator = createProductIterator();
let result = iterator.next();
while (!result.done) {
    console.log(result.value.toString());
    result = iterator.next();
}
```

Генераторы обозначаются звездочкой (*), например, так:

```
...
function* createProductIterator() {
...
}
```

Генераторы используются так же, как итераторы. Среда выполнения JavaScript создает следующую функцию и выполняет функцию-генератор до тех пор, пока не достигнет ключевого слова `yield`, которое предоставляет значение в последовательности. Выполнение функции-генератора продолжается при каждом вызове следующей функции. Когда больше не остается операторов `yield` для выполнения, автоматически создается объект, свойство `done` которого имеет значение `true`.

Генераторы можно использовать с оператором расширения, что позволяет задействовать последовательность в качестве набора параметров функции или для заполнения массива, как показано в листинге 4.18.

Листинг 4.18. Использование оператора расширения в файле index.js

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

function* createProductIterator() {
    yield new Product("Hat", 100);
    yield new Product("Boots", 100);
    yield new Product("Umbrella", 23);
}

[...createProductIterator()].forEach(p => console.log(p.toString()));
```

Оператор `new` в листинге 4.18 использует последовательность значений из генератора для заполнения массива, который затем перебирается с помощью метода `forEach`. Код в листинге 4.18 выдает следующий результат:

```
toString: Name: Hat, Price: 100
toString: Name: Boots, Price: 100
toString: Name: Umbrella, Price: 23
```

4.3.2. Определение итерируемых объектов

Автономные функции для итераторов и генераторов могут быть полезны, но часто требуется, чтобы объект предоставлял последовательность как часть более широкой функциональности. В листинге 4.19 определен объект, который группирует связанные элементы данных и предоставляет генератор для упорядочения этих элементов.

Листинг 4.19. Определение объекта с помощью последовательности в файле index.js

```
class Product {
    constructor(name, price) {
        this.name = name;
```

```

        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

class GiftPack {
    constructor(name, prod1, prod2, prod3) {
        this.name = name;
        this.prod1 = prod1;
        this.prod2 = prod2;
        this.prod3 = prod3;
    }

    getTotalPrice() {
        return [this.prod1, this.prod2, this.prod3]
            .reduce((total, p) => total + p.price, 0);
    }

    *getGenerator() {
        yield this.prod1;
        yield this.prod2;
        yield this.prod3;
    }
}

let winter = new GiftPack("winter", new Product("Hat", 100),
    new Product("Boots", 80), new Product("Gloves", 23));

console.log('Total price: ${ winter.getTotalPrice() }');

[...winter.getGenerator()].forEach(p => console.log('Product: ${ p }'));

```

Класс `GiftPack` служит для отслеживания набора сопутствующих товаров. Один из методов, определенных в `GiftPack`, называется `getGenerator` и представляет собой генератор, выдающий эти товары.

СОВЕТ

Перед именами методов генератора ставится звездочка.

Данный подход работает, однако синтаксис итератора не совсем удобен, поскольку метод `getGenerator` должен быть вызван явно, например, так:

```

...
[...winter.getGenerator()].forEach(p => console.log('Product: ${ p }'));
...

```

Более элегантный подход заключается в использовании специального имени метода для генератора, которое сообщает среде выполнения JavaScript, что метод обеспечивает поддержку итераций по умолчанию для объекта, как показано в листинге 4.20.

Листинг 4.20. Определение метода итератора по умолчанию в файле index.js

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

class GiftPack {
    constructor(name, prod1, prod2, prod3) {
        this.name = name;
        this.prod1 = prod1;
        this.prod2 = prod2;
        this.prod3 = prod3;
    }

    getTotalPrice() {
        return [this.prod1, this.prod2, this.prod3]
            .reduce((total, p) => total + p.price, 0);
    }

    *[Symbol.iterator]() {
        yield this.prod1;
        yield this.prod2;
        yield this.prod3;
    }
}

let winter = new GiftPack("winter", new Product("Hat", 100),
    new Product("Boots", 80), new Product("Gloves", 23));

console.log('Total price: ${ winter.getTotalPrice() }');

[...winter].forEach(p => console.log('Product: ${ p }'));
```

Свойство `Symbol.iterator` используется для обозначения итератора по умолчанию для объекта (не беспокойтесь о свойстве `Symbol` — оно является самым малоиспользуемым примитивом JavaScript и его назначение будет рассмотрено позже). Указание значения `Symbol.iterator` в качестве имени генератора позволяет выполнять итерацию объекта напрямую, например, так:

```
...
[...winter].forEach(p => console.log('Product: ${ p }'));
...
```

Нам больше не нужно вызывать метод для получения генератора, что позволяет получить более чистый и элегантный код.

4.4. КОЛЛЕКЦИИ В JAVASCRIPT

Традиционно управление коллекциями данных в JavaScript осуществляется через объекты и массивы, где объекты используются для хранения данных по ключу, а массивы — по индексу. В JavaScript также предусмотрены специальные объекты-коллекции, которые обеспечивают более структурированный подход к данным, но при этом могут быть менее гибкими. Мы подробно рассмотрим их в следующих разделах.

4.4.1. Хранение данных по ключу с использованием объекта

Объекты могут использоваться как коллекции, где каждое свойство представляет собой пару «ключ/значение», причем ключом является имя свойства, как показано в листинге 4.21.

Листинг 4.21. Использование объекта в качестве коллекции в файле index.js

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

let data = {
    hat: new Product("Hat", 100)
}

data.boots = new Product("Boots", 100);

Object.keys(data).forEach(key => console.log(data[key].toString()));
```

В данном примере объект с именем `data` используется для коллекции объектов `Product`. Новые значения могут быть добавлены в коллекцию путем определения новых свойств, например, так:

```
...
data.boots = new Product("Boots", 100);
...
```

`Object` предоставляет полезные методы для получения набора ключей или значений из объекта, которые обобщены в таблице 4.2.

В листинге 4.21 используется метод `Object.keys` для получения массива, содержащего имена свойств, определенных объектом `data`, а для получения соответствующего значения предназначен метод `forEach` для массивов. Когда имя

свойства присваивается переменной, соответствующее значение может быть получено с помощью квадратных скобок, например, так:

```
...
Object.keys(data).forEach(key => console.log(data[key].toString()));
...

```

Таблица 4.2. Методы объекта для ключей и значений

Название	Описание
Object.keys(object)	Возвращает массив, содержащий имена свойств, определяемых объектом
Object.values(object)	Возвращает массив, содержащий значения свойств, определяемых объектом

Содержимое квадратных скобок оценивается как выражение, а указание имени переменной, например key, возвращает ее значение. Код в листинге 4.21 выдает следующий результат:

```
toString: Name: Hat, Price: 100
toString: Name: Boots, Price: 100
```

4.4.2. Хранение данных по ключу с использованием map

Объекты легко использовать в качестве базовых коллекций, но есть некоторые ограничения, например возможность указывать в качестве ключей только строковые значения. JavaScript также предоставляет коллекцию Map, специально созданную для хранения данных с использованием ключей любого типа, как показано в листинге 4.22.

Листинг 4.22. Использование map в файле index.js в папке primer

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

let data = new Map();
data.set("hat", new Product("Hat", 100));
data.set("boots", new Product("Boots", 100));

[...data.keys()].forEach(key => console.log(data.get(key).toString()));
```

API, предоставляемый `Map`, позволяет хранить и извлекать элементы, а для ключей и значений доступны итераторы. Код в листинге 4.22 выдает тот же результат, что и предыдущий пример. В табл. 4.3 описаны наиболее часто используемые методы.

Таблица 4.3. Полезные методы работы с `Map`

Название	Описание
<code>set(key, value)</code>	Сохраняет значение с указанным ключом
<code>get(key)</code>	Извлекает значение, хранящееся по указанному ключу
<code>keys()</code>	Возвращает итератор для ключей в <code>Map</code>
<code>values()</code>	Возвращает итератор для значений в <code>Map</code>
<code>entries()</code>	Возвращает итератор для пар «ключ/значение» в <code>Map</code> , каждая из которых представляется в виде массива, содержащего ключ и значение. Этот метод является итератором по умолчанию для объектов <code>Map</code>

4.4.3. Использование символов для ключей `Map`

Основное преимущество использования `Map` состоит в том, что в качестве ключа можно использовать любое значение, включая значения `Symbol`. Каждое значение `Symbol` уникально и неизменямо, что делает его идеальным для использования в качестве идентификатора объектов. В листинге 4.23 определена новая коллекция `Map`, в которой в роли ключей используются значения `Symbol`.

ПРИМЕЧАНИЕ

Хотя символьные значения могут быть полезны, с ними сложно работать, поскольку они не представляются в удобочитаемом виде и требуют осторожного создания и обращения. Дополнительную информацию см. по ссылке https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol.

Листинг 4.23. Использование значений `Symbol` в качестве ключей в файле `index.js`

```
class Product {
    constructor(name, price) {
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}

class Supplier {
    constructor(name, productids) {
        this.name = name;
        this.productids = productids;
    }
}
```

```

let acmeProducts = [new Product("Hat", 100), new Product("Boots", 100)];
let zoomProducts = [new Product("Hat", 100), new Product("Boots", 100)];

let products = new Map();
[...acmeProducts, ...zoomProducts].forEach(p => products.set(p.id, p));
let suppliers = new Map();
suppliers.set("acme", new Supplier("Acme Co", acmeProducts.map(p => p.id)));
suppliers.set("zoom",
    new Supplier("Zoom Shoes", zoomProducts.map(p => p.id)));

suppliers.get("acme").productids.forEach(id =>
    console.log(`Name: ${products.get(id).name}`));

```

Преимущество использования значений `Symbol` в качестве ключей состоит в том, что это исключает возможность конфликта двух ключей, который может возникнуть, если ключи берутся из значений характеристик. В предыдущем примере ключом служило значение `Product.name`, а это чревато тем, что два объекта были бы сохранены с одним ключом и один из них заменил бы другой. В данном примере каждый объект `Product` имеет свойство `id`, которому в конструкторе присваивается значение `Symbol`, используемое для хранения объекта в `Map`. Используя `Symbol`, мы можем хранить объекты с одинаковыми свойствами `name` и `price` и без проблем извлекать их. Код, приведенный в листинге 4.23, выдает следующий результат:

```

Name: Hat
Name: Boots

```

4.4.4. Хранение данных по индексу

В главе 3 мы рассмотрели способы хранения данных в массиве. Кроме того, JavaScript предоставляет коллекцию `Set`, которая также хранит данные по индексу, но с дополнительной оптимизацией производительности, и, что особенно удобно, хранит только уникальные значения, как показано в листинге 4.24.

Листинг 4.24. Использование Set в файле index.js

```

class Product {
    constructor(name, price) {
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}

let product = new Product("Hat", 100);

let productArray = [];
let productSet = new Set();

for (let i = 0; i < 5; i++) {
    productArray.push(product);
    productSet.add(product);
}

console.log('Array length: ${productArray.length}');
console.log('Set size: ${productSet.size}');

```

В нашем примере один и тот же объект `Product` пять раз добавляется в массив и коллекцию `Set`, а затем выводится количество элементов в каждом из них, что приводит к следующему результату:

```
Array length: 5
Set size: 1
```

В зависимости от потребностей проекта необходимость разрешить или запретить дублирование значений может определять выбор между массивом и коллекцией `Set`. API, предоставляемый `Set`, обеспечивает функциональность, сравнимую с работой с массивом. В табл. 4.4 описаны наиболее полезные методы.

Таблица 4.4. Полезные методы работы с `Set`

Название	Описание
<code>add(value)</code>	Добавляет значение в <code>Set</code>
<code>entries()</code>	Возвращает итератор для элементов в <code>Set</code> в том порядке, в котором они были добавлены
<code>has(value)</code>	Возвращает значение <code>true</code> , если <code>Set</code> содержит указанное значение
<code>forEach(callback)</code>	Вызывает функцию для каждого значения в <code>Set</code>

4.5. МОДУЛИ

Большинство приложений слишком сложны, чтобы вместить весь код в один файл. JavaScript поддерживает *модули*, что позволяет разбить приложения на управляемые фрагменты. С момента появления JavaScript существовало множество различных подходов к модулям, но сейчас произошла их консолидация и для большинства проектов на JavaScript достаточно двух типов модулей: ECMAScript и CommonJS.

ECMAScript — это официальное название JavaScript, а термин «модуль ECMAScript» относится к недавним дополнениям к спецификации языка JavaScript, описывающим модули. Это «официальная» спецификация модулей, и большинство сред выполнения JavaScript и популярных пакетов сторонних разработчиков поддерживают этот тип модулей, включая среду выполнения Node.js, используемую в данной книге.

CommonJS — это более старая спецификация, которая стала стандартом де-факто, поскольку поддерживалась Node.js до принятия фактического стандарта модулей ECMAScript. В примерах, приводимых в предыдущих изданиях этой книги, использовался формат модуля CommonJS.

СОВЕТ

Для большинства проектов следует использовать модули ECMAScript, если они поддерживаются вашей средой выполнения JavaScript, например Node.js или браузером. Модули ECMAScript не только являются «официальным» стандартом, но и могут импортироваться из модулей CommonJS, что позволяет смешивать и сочетать различные форматы модулей.

4.5.1. Объявление типа модуля

Перед использованием модулей необходимо определиться с форматом: ECMAScript или CommonJS, чтобы среда выполнения Node.js могла обрабатывать файлы. Это можно сделать, создав файлы кода с расширением `.mjs` или `.cjs` (для ECMAScript и CommonJS соответственно). Я предпоготвлю конфигурировать проект с помощью файла `package.json`, как показано в листинге 4.25.

Листинг 4.25. Установка типа модуля в файле `package.json` из папки `primer`

```
{  
  "name": "primer",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "nodemon": "^2.0.20"  
  },  
  "type": "module"  
}
```

Добавление свойства `type` в файл `package.json` устанавливает тип модуля без необходимости использования специальных расширений файлов. Значениями этого свойства являются `module` — для ECMAScript и `commonjs` — для CommonJS.

4.5.2. Создание модуля JavaScript

Каждый модуль JavaScript хранится в соответствующем файле JavaScript. Для создания модуля добавим файл `tax.js` в папку `primer` и поместим в него код, показанный в листинге 4.26.

Листинг 4.26. Содержимое файла `tax.js` из папки `primer`

```
export default function(price) {  
  return Number(price) * 1.2;  
}
```

Функция, определенная в файле `tax.js`, получает значение `price` и применяет 20-процентную ставку налога. Сама функция проста, важны ключевые слова `export` и `default`. Ключевое слово `export` используется для обозначения функций, которые должны быть доступны за пределами модуля. По умолчанию содержимое JavaScript-файла является приватным, и, прежде чем обращаться к нему остальной части приложения, требуется явно указать это с помощью ключевого слова `export`. Ключевое слово `default` используется, когда модуль содержит единственную

функцию, как в примере в листинге 4.26. Вместе ключевые слова `export` и `default` применяются для указания того, что единственная функция в файле `tax.js` доступна для использования с остальной частью приложения.

4.5.3. Использование модуля JavaScript

Для использования модуля JavaScript требуется еще одно ключевое слово — `import`. В листинге 4.27 мы указываем в файле `index.js` ключевое слово `import` для использования функции, определенной в файле `tax.js`.

Листинг 4.27. Использование модуля в файле `index.js` из папки `primer`

```
import calcTax from "./tax.js";

class Product {
    constructor(name, price) {
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}

let product = new Product("Hat", 100);
let taxedPrice = calcTax(product.price);
console.log('Name: ${ product.name }, Taxed Price: ${taxedPrice}');
```

Ключевое слово `import` предназначено для объявления зависимости от модуля. Оно может быть использовано в нескольких вариантах, но именно этот формат будет наиболее часто применяться при работе с модулями, созданными в рамках одного проекта.

За ключевым словом `import` следует идентификатор, представляющий собой имя, под которым функция в модуле будет известна при ее использовании. В нашем примере это `calcTax`. После идентификатора следует ключевое слово `from`, за которым указывается расположение модуля. Важно правильно обозначить расположение, поскольку различные его форматы могут привести к различному поведению, как описано во врезке «О важности расположения модулей» ниже.

В процессе сборки среда выполнения JavaScript обнаружит оператор `import` и загрузит содержимое файла `tax.js`. Идентификатор, используемый в операторе `import`, может быть задействован для доступа к функции в модуле, точно так же, как если бы она была определена локально.

```
...
let taxedPrice = calcTax(product.price);
...
```

При выполнении этого кода значение, присвоенное переменной `taxedPrice`, вычисляется с помощью функции, определенной в файле `tax.js`, и выдает следующий результат:

`Name: Hat, Taxed Price: 120`

О ВАЖНОСТИ РАСПОЛОЖЕНИЯ МОДУЛЕЙ

Расположение модуля определяет, где среда выполнения JavaScript будет искать файл с кодом модуля. Для модулей, определенных в проекте, расположение задается в виде относительного пути, который начинается с одной или двух точек, указывающих, что путь является относительным к текущему файлу или родительскому каталогу текущего файла. В листинге 4.27 путь к модулю начинается с точки.

```
...  
import calcTax from "./tax.js";  
...
```

Это расположение сообщает инструментам сборки, что существует зависимость от модуля `tax`, который можно найти в той же папке, что и файл, содержащий оператор `import`.

Если вы опустите начальную точку или точки, оператор `import` объявит зависимость от модуля, которого нет в локальном проекте. Места поиска модуля будут различаться в зависимости от фреймворка приложения и используемых вами инструментов сборки, но чаще всего поиск ведется в `node_modules`, куда пакеты загружаются во время установки проекта. Это расположение используется для доступа к функциям, предоставляемым сторонними пакетами. Примеры использования модулей из сторонних пакетов рассмотрены в части III, но в качестве краткой справки приведу оператор `import` из главы, посвященной разработке с помощью React:

```
...  
import React, { Component } from "react";  
...
```

Расположение этого оператора `import` не начинается с точки и будет интерпретироваться как зависимость от модуля `react` в папке `node_modules` проекта, который является пакетом, предоставляющим основные функции React-приложения.

4.5.4. Экспорт именованных функций из модуля

Модуль может присваивать имена экспортимым им функциям. В листинге 4.28 мы прибегнем именно к такому подходу и присвоим имя функции, экспортимой модулем `tax`.

Листинг 4.28. Экспорт именованной функции в файле `tax.js` из папки `primer`

```
export function calculateTax(price) {  
    return Number(price) * 1.2;  
}
```

Функция предоставляет ту же возможность, но экспортируется под именем `calculateTax` и больше не использует ключевое слово `default`. В листинге 4.29 мы импортируем функцию под новым именем в файл `index.js`.

Листинг 4.29. Импорт именованной функции в файл `index.js` из папки `primer`

```
import { calculateTax } from "./tax.js";

class Product {
    constructor(name, price) {
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}

let product = new Product("Hat", 100);
let taxedPrice = calculateTax(product.price);
console.log('Name: ${ product.name }, Taxed Price: ${taxedPrice}');
```

Имя импортируемой функции указывается в фигурных скобках (`{}`) и используется под этим именем в коде. Модуль может экспортировать стандартные и именованные функции, как показано в листинге 4.30.

Листинг 4.30. Экспорт именованных функций и функций по умолчанию в файле `tax.js`

```
export function calculateTax(price) {
    return Number(price) * 1.2;
}

export default function calcTaxAndSum(...prices) {
    return prices.reduce((total, p) => total += calculateTax(p), 0);
}
```

Новая функция экспортируется с помощью ключевого слова `default`. В листинге 4.31 мы импортируем новую функцию в качестве экспорта по умолчанию из модуля.

Листинг 4.31. Импорт функции по умолчанию в файл `index.js`

```
import calcTaxAndSum, { calculateTax } from "./tax.js";

class Product {
    constructor(name, price) {
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}

let product = new Product("Hat", 100);
let taxedPrice = calculateTax(product.price);
console.log('Name: ${ product.name }, Taxed Price: ${taxedPrice}');

let products = [new Product("Gloves", 23), new Product("Boots", 100)];
let totalPrice = calcTaxAndSum(...products.map(p => p.price));
console.log('Total Price: ${totalPrice.toFixed(2)}');
```

Это общий шаблон для таких фреймворков веб-приложений, как React, где основные функции предоставляются путем экспорта модуля по умолчанию, а дополнительные функции доступны в виде именованных экспортов. Код из листинга 4.31 выдает следующий результат:

```
Name: Hat, Taxed Price: 120
Total Price: 147.60
```

4.5.5. Определение нескольких именованных функций в модуле

Модули могут содержать более одной именованной функции или значения, что удобно для группировки связанных функций. Для демонстрации этого примера в папку primer был добавлен файл utils.js с кодом, показанным в листинге 4.32.

Листинг 4.32. Содержимое файла utils.js из папки primer

```
import { calculateTax } from "./tax.js";

export function printDetails(product) {
    let taxedPrice = calculateTax(product.price);
    console.log('Name: ${product.name}, Taxed Price: ${taxedPrice}');
}

export function applyDiscount(product, discount = 5) {
    product.price = product.price - discount;
}
```

В данном модуле определены две функции, к которым применено ключевое слово `export`. В отличие от предыдущего примера ключевое слово `default` не используется и каждая функция имеет собственное имя. При импорте из модуля, содержащего несколько функций, имена используемых функций указываются в виде списка, разделенного запятыми внутри фигурных скобок, как показано в листинге 4.33.

Листинг 4.33. Импорт именованных функций в файл index.js в папке primer

```
import calcTaxAndSum, { calculateTax } from "./tax.js";
import { printDetails, applyDiscount } from "./utils.js";

class Product {
    constructor(name, price) {
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}

let product = new Product("Hat", 100);
applyDiscount(product, 10);
//let taxedPrice = calculateTax(product.price);
printDetails(product);

let products = [new Product("Gloves", 23), new Product("Boots", 100)];
let totalPrice = calcTaxAndSum(...products.map(p => p.price));
console.log('Total Price: ${totalPrice.toFixed(2)}');
```

Здесь задействуется оператор `import`, который включает в себя список функций в фигурных скобках. Это позволяет импортировать только те функции, которые фактически используются в коде, не загромождая его неиспользуемыми функциями. Код в листинге 4.33 выдает следующий результат:

```
Name: Hat, Taxed Price: 112  
Total Price: 147.60
```

РЕЗЮМЕ

В данной главе были рассмотрены ключевые особенности JavaScript, которые служат фундаментом для эффективной разработки на TypeScript. Важно понимать, как работать с объектами, последовательностями значений, коллекциями и модулями. Все это возможности JavaScript, но, как вы узнаете, их понимание закладывает основу для эффективной разработки на TypeScript. Объекты JavaScript имеют прототип, от которого они наследуют свойства и методы.

- Создание объектов может осуществляться как с помощью литералов, так с использованием функций-конструкторов.
- Классы в JavaScript предоставляют согласованный шаблон для создания объектов.
- Классы в JavaScript поддерживают приватные поля, свойства и методы.
- Итераторы и генераторы служат для создания последовательности значений.
- В JavaScript для простых коллекций часто используются объекты и массивы, а также встроенная коллекция `Map`.
- Модули позволяют эффективно структурировать проект с помощью форматов модулей ECMAScript или CommonJS.

Следующая глава будет посвящена компилятору TypeScript, лежащему в основе всех возможностей, которые TypeScript предоставляет разработчикам.

5

Компилятор *TypeScript*

В этой главе

- ✓ Установка пакета TypeScript с помощью менеджера пакетов Node.
- ✓ Создание файла конфигурации для компилятора TypeScript.
- ✓ Использование компилятора TypeScript для генерации кода JavaScript.
- ✓ Выбор версии языка JavaScript, на которую ориентирован компилятор TypeScript.
- ✓ Определение формата JavaScript-модуля, используемого компилятором TypeScript.

В этой главе вы узнаете, как использовать компилятор TypeScript. Он отвечает за преобразование кода TypeScript в JavaScript, который может выполняться браузерами или средой исполнения Node.js. Также будут рассмотрены параметры конфигурации компилятора, наиболее полезные для разработки на TypeScript, в том числе те, что используются фреймворками веб-приложений, о которых пойдет речь в части III.

5.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

Для того чтобы приступить к работе с материалом из этой главы, откройте командную строку, перейдите в удобное место и создайте папку `tools`. Затем выполните команды, показанные в листинге 5.1, чтобы перейти в каталог `tools` и указать менеджер пакетов Node (NPM) создать файл с именем `package.json`. Этот файл будет

использоваться для отслеживания пакетов, добавленных в проект, как описано в разделе 5.3 далее.

Листинг 5.1. Создание файла package.json

```
cd tools
npm init --yes
```

С помощью командной строки выполните команды, приведенные в листинге 5.2, чтобы установить в папку tools пакет, необходимый для данной главы.

Листинг 5.2. Добавление пакетов с помощью менеджера пакетов Node

```
npm install --save-dev typescript@5.0.2
npm install --save-dev tsc-watch@6.0.0
```

Аргумент `install` указывает NPM на загрузку и добавление пакета в текущую папку, а `--save-dev` — на то, что эти пакеты предназначены для использования в разработке, а не являются частью приложения. Последний аргумент — это имя пакета, за которым следует символ @ с требуемой версией пакета.

ПРИМЕЧАНИЕ

Важно использовать версии, указанные в книге; использование других версий может привести к неожиданному поведению или ошибкам.

Чтобы создать конфигурационный файл для компилятора TypeScript, добавьте файл с именем `tsconfig.json` в папку `tools` с содержимым, показанным в листинге 5.3.

Листинг 5.3. Содержимое файла tsconfig.json из папки tools

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Для завершения настройки создайте папку `tools/src` и добавьте в нее файл с именем `index.ts`, содержащий код, приведенный в листинге 5.4.

Листинг 5.4. Содержимое файла index.ts из папки src

```
function printMessage(msg: string): void {
  console.log('Message: ${ msg }');
}

printMessage("Hello, TypeScript");
```

Чтобы скомпилировать код TypeScript, запустите в каталоге `tools` команду, показанную в листинге 5.5.

Листинг 5.5. Компиляция кода TypeScript`tsc`

Чтобы выполнить скомпилированный код, наберите команду, показанную в листинге 5.6, в папке tools.

Листинг 5.6. Запуск скомпилированного кода`node dist/index.js`

Если проект успешно настроен, то в консоли появится следующий вывод:

Message: Hello, TypeScript

5.2. СТРУКТУРА ПРОЕКТА

Структура примера проекта соответствует структуре большинства разработок на JavaScript и TypeScript, с некоторыми вариациями, зависящими от используемого фреймворка, например React или Angular. На рис. 5.1 показано содержимое папки tools.

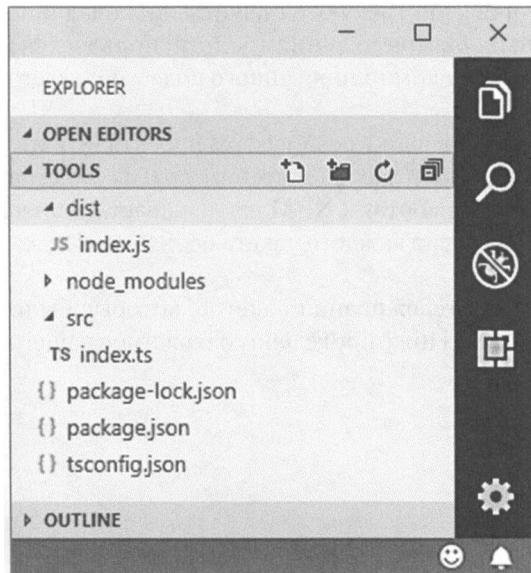


Рис. 5.1. Содержимое папки tools

Здесь видно, как папка проекта отображается в Visual Studio Code – редакторе, который я использую на протяжении всей книги. В табл. 5.1 описаны все элементы проекта. В последующих разделах вы подробнее познакомитесь с наиболее важными элементами.

Таблица 5.1. Файлы и папки проекта

Название	Описание
dist	Папка, содержащая выходные данные компилятора
node_modules	Папка, содержащая пакеты, необходимые приложению и средствам разработки, как описано в разделе 5.3
src	Папка, содержащая файлы исходного кода, которые будут скомпилированы компилятором TypeScript
package.json	Папка, содержащая набор зависимостей пакетов верхнего уровня для проекта, как описано в разделе 5.3
package-lock.json	Файл, содержащий полный список зависимостей пакетов для проекта
tsconfig.json	Файл, содержащий настройки конфигурации для компилятора TypeScript

5.3. МЕНЕДЖЕР ПАКЕТОВ NODE

Разработка на TypeScript и JavaScript тесно связана с развитой экосистемой пакетов. Для большинства TypeScript-проектов требуются пакеты, включающие в себя компилятор TypeScript, фреймворк приложения (если используется) и инструменты, необходимые для упаковки скомпилированного кода с целью его распространения и выполнения.

Для загрузки и добавления этих пакетов в папку `node_modules` проекта используется NPM. Каждый пакет декларирует набор зависимостей от других пакетов и указывает версии, с которыми он может работать. NPM отслеживает данную цепочку зависимостей, определяя, какая версия каждого пакета необходима, и загружает все требуемые файлы.

Файл `package.json` используется для отслеживания пакетов, которые были добавлены с помощью команды `npm install`. Ниже приведено содержимое нашего файла `package.json`:

```
{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^6.0.0",
    "typescript": "^5.0.2"
  }
}
```

Основное содержимое файла было сгенерировано командой `npm init` в листинге 5.1 и затем изменялось при каждом использовании команды `npm install` в листинге 5.2. Пакеты разделяются на те, которые используются в процессе разработки, и те, которые являются частью самого приложения. Пакеты, используемые во время разработки, устанавливаются с помощью аргумента `--save-dev` и записываются в раздел `devDependencies` файла `package.json`. Пакеты, которые входят в состав приложения, устанавливаются без аргумента `--save-dev` и сохраняются в разделе `dependencies`. В листинге 5.2 были установлены только пакеты для разработки, поэтому все они находятся в разделе `devDependencies`, а файл `package.json` вообще не содержит раздела `dependencies`. В последующих примерах книги пакеты будут добавляться в раздел `dependencies`, но в данной главе основное внимание уделяется инструментам, используемым для разработки TypeScript. В табл. 5.2 описаны все пакеты, которые были добавлены в проект.

ГЛОБАЛЬНЫЕ И ЛОКАЛЬНЫЕ ПАКЕТЫ

Менеджеры пакетов могут устанавливать пакеты так, чтобы они были предназначены только для одного проекта (так называемая локальная установка) или чтобы к ним можно было получить доступ из любого места (так называемая глобальная установка). В главе 2 пакет `typescript` был установлен глобально, что позволяет использовать команду `tsc` для компиляции кода в любом месте. В листинге 5.2 тот же пакет установлен локально, хотя его функциональность уже доступна. Это сделано для того, чтобы другие пакеты в том же проекте могли получить доступ к функциональности, предоставляемой компилятором TypeScript.

Таблица 5.2. Пакеты, добавленные в пример проекта

Название	Описание
<code>tsc-watch</code>	Отслеживает папку с исходным кодом, при изменении запускает компилятор TypeScript и выполняет скомпилированный JavaScript-код
<code>typescript</code>	Содержит компилятор TypeScript и его вспомогательные инструменты

Для каждого пакета в файле `package.json` указываются допустимые номера версий в формате, описанном в табл. 5.3.

Таблица 5.3. Система нумерации версий пакетов

Формат	Описание
<code>5.0.2</code>	Требует точного совпадения с указанной версией (5.0.2)
<code>*</code>	Позволяет установить любую доступную версию пакета
<code>>5.0.2</code>	Допускает использование версий пакета начиная с более новой, чем указанная (>), или начиная с указанной и новее (>=)
<code>>=5.0.2</code>	

Таблица 5.3 (продолжение)

Формат	Описание
<5.0.2 ≤5.0.2	Допускает использование версий пакета меньше, чем указанная (<), или меньше или равной указанной (≤=)
~5.0.2	Позволяет установить версии, даже если номер последнего уровня исправления (последний из трех номеров версий) не совпадает. Например, если указать ~5.0.2, будут разрешены версии 5.0.3 и 5.0.4, но не 5.1.0 (которая будет новым минорным выпуском)
^5.0.2	Позволяет установить версии, даже если номер минорного уровня (второй из трех номеров версии) или номер исправления не совпадают. Например, указание ^5.0.2 разрешит использовать версии 5.0.3 и 5.1.0, но не версию 6.0.0

NPM – сложный инструмент, и понимание его использования является важной частью разработки на JavaScript и TypeScript. В табл. 5.4 приведены некоторые команды NPM, которые могут вам пригодиться во время разработки. Все эти команды следует выполнять в папке проекта, содержащей файл `package.json`.

Таблица 5.4. Полезные команды NPM

Команда	Описание
<code>npm install</code>	Локально устанавливает пакеты, перечисленные в файле <code>package.json</code>
<code>npm install package@version</code>	Локально устанавливает указанную версию пакета и обновляет файл <code>package.json</code> , добавляя пакет в раздел <code>dependencies</code>
<code>npm install --save-dev package@version</code>	Локально устанавливает указанную версию пакета и обновляет файл <code>package.json</code> , добавляя пакет в раздел <code>devDependencies</code>
<code>npm install --global package@version</code>	Глобально устанавливает указанную версию пакета
<code>npm list</code>	Выводит список всех локальных пакетов и их зависимостей
<code>npm run</code>	Выполняет один из скриптов, определенных в файле <code>package.json</code>
<code>npx package</code>	Запускает код, содержащийся в пакете

Папка `node_modules` обычно исключается из контроля версий, поскольку содержит большое количество файлов, а пакеты могут содержать компоненты, специфичные для конкретной платформы, которые не будут работать при проверке проекта на новой машине. Вместо этого для создания новой папки `node_modules` и установки необходимых пакетов следует использовать команду `npm install`.

При таком подходе каждый раз при выполнении команды `npm install` вы рискуете получить разный набор пакетов, поскольку зависимости могут быть выражены в виде диапазона версий, как описано в табл. 5.4. Для обеспечения согласованности

NPM создает файл `package-lock.json`, который содержит полный список пакетов, установленных в папке `node_module`, с указанием конкретных версий. Файл `package-lock.json` обновляется NPM при внесении изменений в пакеты проекта, а содержащиеся в нем версии используются командой `npm install`.

ПРИМЕЧАНИЕ

Файлы `package.json` и `package-lock.json` должны быть добавлены в систему контроля версий, чтобы все члены команды разработчиков получали одни и те же пакеты. При извлечении обновлений из репозитория обязательно выполняйте команду `npm install`, чтобы получить все новые пакеты, добавленные другим разработчиком.

5.4. ФАЙЛ КОНФИГУРАЦИИ КОМПИЛЯТОРА

Компилятор TypeScript — `tsc` — отвечает за компиляцию файлов TypeScript. Именно компилятор отвечает за реализацию возможностей TypeScript, включая статическую типизацию, и в результате получается чистый JavaScript, из которого удалены ключевые слова и выражения TypeScript.

Компилятор имеет множество опций конфигурации, о которых будет рассказано далее в этой главе. Файл конфигурации используется для переопределения настроек по умолчанию и обеспечивает постоянное использование последовательной конфигурации. Имя конфигурационного файла — `tsconfig.json`, который был создан в листинге 5.3 со следующим содержимым:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Файл `tsconfig.json` может содержать несколько параметров конфигурации верхнего уровня, как описано в табл. 5.5, хотя файл, используемый в примере проекта, содержит только параметры `compilerOptions`, которые описаны в разделе 5.9.

Таблица 5.5. Конфигурационные параметры верхнего уровня файла `tsconfig.json`

Название	Описание
<code>compilerOptions</code>	В этом разделе группируются параметры, которые будет использовать компилятор
<code>files</code>	Этот параметр определяет файлы, которые будут скомпилированы, что переопределяет поведение по умолчанию, когда компилятор ищет файлы для компиляции

Таблица 5.5 (продолжение)

Название	Описание
include	Этот параметр используется для выбора файлов для компиляции по шаблону. Если он не указан, то будут выбраны файлы с расширениями <code>.ts</code> , <code>.tsx</code> и <code>.d.ts</code> (файлы <code>TSX</code> и файлы с расширением <code>.d.ts</code> описаны в части III)
exclude	Этот параметр используется для исключения файлов из компиляции по шаблону
compileOnSave	При установке значения <code>true</code> данный параметр указывает редактору кода, что он должен запускать компилятор при каждом сохранении файла. Подобная возможность поддерживается не всеми редакторами, и более полезной альтернативой является функция <code>watch</code> , описанная в следующем разделе

Опции `files`, `include` и `exclude` полезны при нестандартной структуре проекта, например при интеграции TypeScript в проект, содержащий другой фреймворк или инструментарий с конфликтующим набором файлов. Посмотреть набор файлов, доступных для компиляции, можно с помощью параметра `listFiles`, который может быть задан в секции `compilerOptions` файла `tsconfig.json` или указан в командной строке. В качестве примера выполним команду, показанную в листинге 5.7, в папке `tools`, чтобы увидеть файлы, выбранные конфигурацией компилятора.

Листинг 5.7. Отображение списка файлов для компиляции

```
tsc --listFiles
```

Команда выведет длинный список файлов, которые нашел компилятор, включая декларации типов:

```
...
C:/npm/node_modules/typescript/lib/lib.es5.d.ts
C:/npm/node_modules/typescript/lib/lib.es2015.d.ts
C:/npm/node_modules/typescript/lib/lib.es2016.d.ts
C:/npm/node_modules/typescript/lib/lib.es2017.d.ts
C:/npm/node_modules/typescript/lib/lib.es2018.d.ts
C:/npm/node_modules/typescript/lib/lib.es2020.d.ts
C:/npm/node_modules/typescript/lib/lib.es2021.d.ts
C:/npm/node_modules/typescript/lib/lib.es2022.d.ts
...
```

Как объяснялось в главе 2, декларации типов описывают типы данных, используемые в коде JavaScript, чтобы его можно было безопасно использовать в приложении TypeScript. Пакет TypeScript включает в себя декларации типов для различных версий языка JavaScript, а также для API, доступных в Node.js и браузерах. Более подробно декларации типов описаны в части III, а конкретные файлы — в разделе 5.6.

ПРИМЕЧАНИЕ

Пути к файлам деклараций типов находятся вне проекта, поскольку команда `tsc` запускает компилятор TypeScript из пакета, установленного глобально в главе 2. Этот же пакет был установлен локально в папке `node_modules` и используется в процессе разработки, о чем будет сказано в следующем разделе. Если необходимо запустить компилятор из пакета, установленного локально в проекте, воспользуйтесь командой `prx`. Например, `prx tsc listFiles` действует аналогично команде из листинга 5.7, но использует локальный пакет.

Этот файл появляется в конце списка, формируемого опцией `listFile`:

```
...
C:/tools/src/index.ts
...
```

В процессе обнаружения компилятор TypeScript ищет файлы TypeScript в месте, указанном параметром `rootDir` в файле `tsconfig.json`. Компилятор просматривает папку `src` и находит файл `index.ts`.

5.5. КОМПИЛЯЦИЯ TYPESCRIPT-КОДА

Компилятор проверяет код TypeScript на предмет реализации такой функциональности, как статические типы, и генерирует чистый JavaScript-код, из которого удалены дополнения TypeScript. Компилятор можно запустить непосредственно из консоли, и он обрабатывает все файлы, указанные в опции `listfile`. Для запуска компилятора выполните команду, показанную в листинге 5.8, в папке `tools`.

Листинг 5.8. Запуск компилятора

```
tsc
```

В проекте есть только один файл TypeScript — `src/index.ts`, и конфигурационные настройки в файле `tsconfig.json` указывают компилятору, что результаты компиляции нужно поместить в папку `dist`. Если изучить содержимое данной папки, то можно увидеть, что в ней находится файл `index.js` со следующим наполнением:

```
function printMessage(msg) {
    console.log('Message: ${msg}');
}
printMessage("Hello, TypeScript");
```

Файл `index.js` содержит скомпилированный код из файла `index.ts` в папке `src`, но без дополнительной информации о типе для функции `printMessage`. Взаимосвязь между кодом TypeScript и кодом JavaScript, сгенерированным компилятором, не всегда будет такой прямой, особенно если компилятору даны указания использовать другую версию JavaScript, как описано в разделе 5.6.

ВНИМАНИЕ

Не редактируйте файлы JavaScript в папке `dist`, поскольку при следующем запуске компилятора TypeScript изменения будут перезаписаны. Изменения должны вноситься только в файлы TypeScript.

5.5.1. Об ошибках компилятора

Компилятор TypeScript проверяет компилируемый код на соответствие спецификации языка JavaScript и применяет такие функции TypeScript, как статические типы и ключевые слова управления доступом. Чтобы показать простой пример ошибки компилятора, в листинг 5.9 добавлен оператор, использующий неверный тип данных для вызова функции `printMessage`.

Листинг 5.9. Создание несоответствия типов в файле `index.ts` из папки `src`

```
function printMessage(msg: string): void {
    console.log('Message: ${ msg }');
}

printMessage("Hello, TypeScript");
printMessage(100);
```

Для запуска компилятора в папке `tools` выполните команду, показанную в листинге 5.10.

СОВЕТ

Функция `printMessage` указывает тип данных, который она ожидает получить в параметре `msg` с помощью аннотации типа, которая описана в главе 7. Сейчас достаточно знать, что вызов функции `printMessage` с числовым значением является ошибкой TypeScript.

Листинг 5.10. Запуск компилятора

```
tsc
```

Компилятор обнаруживает, что тип аргумента в новом операторе — `number`, а не `string`, как ожидает функция `printMessage`, и выдает следующее сообщение:

```
src/index.ts:6:14 - error TS2345: Argument of type 'number' is not
  assignable to parameter of type 'string'.
```

```
6 printMessage(100);
          ~~~
Found 1 error in src/index.ts:6
```

Фактически компилятор TypeScript работает так же, как и любой другой компилятор. Но есть одно отличие, которое может смутить новичков: по умолчанию компилятор продолжает генерацию JavaScript-кода даже при возникновении ошибки. Если изучить содержимое файла `index.js` в папке `dist`, то можно увидеть следующее:

```
function printMessage(msg) {
    console.log('Message: ${msg}');
}
printMessage("Hello, TypeScript");
printMessage(100);
```

Такое странное поведение может вызвать проблемы с конвейерами инструментов, выполняющими или дополнительно обрабатывающими JavaScript-код,

выданный компилятором TypeScript. К счастью, это поведение можно отключить, установив в файле `tsconfig.json` параметр `noEmitOnError` в значение `true`, как показано в листинге 5.11.

Листинг 5.11. Изменение конфигурации в файле `tsconfig.json` из папки `tools`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true
  }
}
```

При подобных настройках компилятор будет генерировать вывод только в том случае, если в коде JavaScript не обнаружено ошибок.

5.5.2. Режим отслеживания и выполнение скомпилированного кода

Используя режим отслеживания изменений компилятора TypeScript, вы можете избежать ручного запуска компилятора после каждого изменения кода, что часто бывает утомительным. Выполните команду, показанную в листинге 5.12, для запуска компилятора в режиме отслеживания.

Листинг 5.12. Запуск компилятора в режиме отслеживания

```
tsc --watch
```

Компилятор запустится, сообщит о той же ошибке, что показана в предыдущем разделе, а затем начнет отслеживать проект на предмет изменений кода. Чтобы запустить компиляцию, закомментируйте проблемную строку, добавленную в файл `index.ts`, как показано в листинге 5.13.

ВНИМАНИЕ

Если при запуске компилятора TypeScript в режиме отслеживания вы можете столкнуться с ошибкой в Node.js, такой как `Check failed: U_SUCCESS(status)`, возможно, необходимо обновить Node.js до последней версии. В противном случае просто перейдите к следующему разделу, поскольку режим отслеживания используется только в этой части главы и больше в примерах книги не задействован.

Листинг 5.13. Комментирование утверждения в файле `index.ts` из папки `src`

```
function printMessage(msg: string): void {
  console.log('Message: ${ msg }');
}

printMessage("Hello, TypeScript");
//printMessage(100);
```

После сохранения изменений автоматически запускается компилятор. Ошибок в коде нет, и компилятор выведет следующий результат:

```
[6:37:35 AM] File change detected. Starting incremental compilation...
[6:37:35 AM] Found 0 errors. Watching for file changes.
```

Для запуска скомпилированного кода откройте второе окно консоли, перейдите в папку `tools` и выполните команду, показанную в листинге 5.14.

Листинг 5.14. Выполнение скомпилированного кода

```
node dist/index.js
```

Среда выполнения Node.js выполнит утверждения из файла `index.js` в каталоге `dist` и выдаст следующий результат:

```
Message: Hello, TypeScript
```

Автоматическое выполнение кода после компиляции

Режим отслеживания не включает автоматическое выполнение скомпилированного кода. Иногда у некоторых разработчиков возникает соблазн объединить данный режим с инструментом, выполняющим команду при обнаружении изменения, однако это может быть затруднительно, поскольку не все файлы JavaScript записываются одновременно и нет надежного способа точно определить, когда компиляция завершится. Если вы используете фреймворк для веб-разработки, например React или Angular, компилятор TypeScript интегрируется в более крупный набор инструментов, который автоматически выполняет скомпилированный код, как будет показано в части III. Для автономных проектов существуют пакеты с открытым исходным кодом, которые расширяют функционал, предоставляемый компилятором, и предлагают дополнительные возможности. Один из таких пакетов — `tswatch`, который был установлен ранее (см. листинг 5.2). Он запускает компилятор в режиме отслеживания, наблюдает за его выходными данными и выполняет команды в зависимости от результатов компиляции. Для запуска пакета `tswatch` в папке `tools` выполните команду, показанную в листинге 5.15.

Листинг 5.15. Запуск команды package

```
npx tsc-watch --onsuccess "node dist/index.js"
```

Аргумент `onsuccess` задает команду, которая выполняется при успешной и безошибочной компиляции. Внесите изменения в файл `index.ts`, как показано в листинге 5.16, чтобы запустить компиляцию и выполнить ее результат.

СОВЕТ

Дополнительную информацию о других опциях, предоставляемых пакетом `tswatch`, можно найти по ссылке <https://github.com/gilamran/tsc-watch>.

Листинг 5.16. Внесение изменений в файл index.ts из папки src

```
function printMessage(msg: string): void {
    console.log('Message: ${ msg }');
}

printMessage("Hello, TypeScript");
printMessage("It is sunny today");
```

После сохранения изменений компилятор TypeScript обнаружит их и скомпилирует файл TypeScript. Затем пакет `tswatch`, увидев, что компилятор не сообщает об ошибках, запустит команду, которая выполнит скомпилированный код, выдав следующий результат:

```
7:20:25 AM - File change detected. Starting incremental compilation...
7:20:25 AM - Found 0 errors. Watching for file changes.
Message: Hello, TypeScript
Message: It is sunny today
```

ПРИМЕЧАНИЕ

Компилятор TypeScript также предоставляет API, который можно использовать для создания пользовательских инструментов, что может быть полезно, если необходимо интегрировать компилятор в сложный рабочий процесс. Несмотря на то что Microsoft не предоставляет подробной документации по API, некоторые заметки и примеры можно найти на сайте <https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API>.

Запуск компилятора с помощью npm

Компилятор TypeScript не реагирует на изменения во всех своих конфигурационных параметрах, и в некоторых случаях может потребоваться остановить и снова запустить компилятор. Вместо ввода команды в листинге 5.16 более надежным способом является использование секции `scripts` файла `package.json`, как показано в листинге 5.17.

Листинг 5.17. Добавление в раздел scripts файла package.json из папки tools

```
{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onsuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^6.0.0",
    "typescript": "^5.0.2"
  }
}
```

Необходимо позаботиться о том, чтобы экранировать символы кавычек, необходимые для аргумента `onsuccess`. Сохраните изменения в файле `package.json` и выполните команду, показанную в листинге 5.18, в папке `tools`.

Листинг 5.18. Запуск компилятора

```
npm start
```

Эффект тот же, но теперь компилятор можно запустить без необходимости запоминать комбинацию имен пакетов и файлов, что в реальных проектах может быть весьма сложным.

5.6. ФУНКЦИЯ ВЫБОРА ЦЕЛЕВОЙ ВЕРСИИ

TypeScript опирается на самые последние версии языка JavaScript, в которых появились классы. Чтобы упростить внедрение TypeScript, компилятор может генерировать JavaScript-код, ориентированный на старые версии языка JavaScript. Это означает, что вы можете использовать последние возможности при разработке, но ваш код все равно будет выполняться в более старых средах выполнения, например устаревшими браузерами.

Версия языка JavaScript, на которую ориентируется компилятор, задается параметром `target` в файле `tsconfig.json`, как показано в листинге 5.19.

Листинг 5.19. Выбор целевой версии в файле `tsconfig.json` из папки `tools`

```
{
  "compilerOptions": {
    "target": "ES5",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true
  }
}
```

Целевая версия JavaScript выбирается из списка, приведенного в табл. 5.6.

ПРИМЕЧАНИЕ

ES в этих настройках означает ECMAScript – стандарт, определяющий функциональность, реализованную в языке JavaScript. История JavaScript и ECMAScript весьма долгая, мучительная и совсем не интересная. В контексте разработки TypeScript можно рассматривать JavaScript и ECMAScript как одно и то же (именно так я подходил к их рассмотрению в книге). Если вас интересует более подробная информация, рекомендую заглянуть сюда: <https://en.wikipedia.org/wiki/ECMAScript>.

Таблица 5.6. Значения для указания целевой версии

Название	Описание
ES3	Ориентируется на третью редакцию спецификации языка, которая была определена в декабре 1999 года и считается базовой. Это значение по умолчанию, когда целевая версия <code>target</code> не определена
ES5	Предназначено для пятого издания спецификации языка, которое было определено в декабре 2009 года, и ориентируется на согласованность (четвертого издания не было)
ES6	Ориентируется на шестое издание спецификации языка и включает функциональность, необходимую для создания сложных приложений, таких как классы, модули, стрелочные функции и промисы (promises)
ES2015	Эквивалентно ES6
ES2016	Ориентируется на седьмое издание спецификации языка, в котором введен метод <code>includes</code> для массивов и оператор возведения в степень

Название	Описание
ES2017	Предназначено для восьмого издания спецификации языка, в котором появились функции проверки объектов и новые ключевые слова для асинхронных операций
ES2018	Ориентируется на девятое издание спецификации языка, в котором введены операторы расширения spread и rest, а также усовершенствования для работы со строками и асинхронными операциями
ES2019	Предназначено для десятого издания спецификации языка, которое включает в себя новые возможности работы с массивами, изменения в обработке ошибок и улучшения в форматировании JSON
ES2020	Ориентируется на 11-е издание спецификации языка, которое включает поддержку оператора нулевого слияния (? ?), optional chaining, цепочки и динамической загрузки модулей
ES2021	Ориентируется на 12-е издание спецификации языка, которое включает в себя поддержку новых логических операторов присваивания, «слабых» ссылок на память, а также разделителей для числовых литералов
ES2022	Фокусируется на 13-м издании спецификации языка, которое включает поддержку приватных членов класса
esNext	Относится к функциям, которые предполагается включить в следующую редакцию спецификации. Конкретные функции, поддерживаемые компилятором TypeScript, могут меняться в разных версиях. Это расширенная настройка, которую следует использовать с осторожностью

Более ранним версиям стандарта ECMAScript были присвоены номера, но последние версии обозначаются годом, в котором они были приняты. Это изменение произошло на этапе определения ES6, поэтому он известен и как ES6, и как ES2015. Наибольшие изменения в язык были привнесены именно с этой версией, которая считается началом «современного» JavaScript. Выпуск ES6 ознаменовал переход на ежегодное обновление спецификации языка, поэтому выпуски с 2016 по 2022 год содержат лишь небольшое количество изменений.

В параметре `target` в листинге 5.19 установлено значение `es5`, а это означает, что ключевое слово `let` и стрелочные функции поддерживаться не будут. Чтобы увидеть, как компилятор использует эти возможности, внесите в файл `index.ts` изменения, показанные в листинге 5.20.

Листинг 5.20. Использование современных возможностей в файле `index.ts` из папки `src`

```
let printMessage = (msg: string)
  : void => console.log('Message: ${ msg }');

let message = ("Hello, TypeScript");
printMessage(message);
```

После сохранения изменений в файле код будет скомпилирован и выполнен. В результате в файле `index.js` в папке `dist` будет содержаться следующий генерированный компилятором JavaScript-код:

```
var printMessage = function (msg) {
    return console.log("Message: ".concat(msg));
};
var message = ("Hello, TypeScript");
printMessage(message);
```

Ключевое слово `let` заменено `var`, а стрелочная функция — традиционной. Код достигает того же эффекта, что и при работе с более новой версией JavaScript, и выводит следующее:

```
Message: Hello, TypeScript
```

5.7. УСТАНОВКА ФАЙЛОВ БИБЛИОТЕК ДЛЯ КОМПИЛЯЦИИ

В выводе с параметром компилятора `listFiles` показаны файлы, которые обнаруживает компилятор, и включен ряд файлов объявлений типов. Эти файлы предоставляют компилятору информацию о типах, доступных в различных версиях JavaScript, и о возможностях, предоставляемых приложениям, работающим в браузере, которые могут создавать и управлять содержимым HTML с помощью API объектной модели документа (DOM).

Компилятор по умолчанию использует необходимую ему информацию о типе на основе свойства `target`, то есть при использовании возможностей более поздних версий JavaScript будут генерироваться ошибки, как показано в листинге 5.21.

Листинг 5.21. Использование современной функции JavaScript в файле `index.ts` из папки `src`

```
let printMessage = (msg: string)
  : void => console.log('Message: ${ msg }');

let message = ("Hello, TypeScript");
printMessage(message);

let data = new Map();
data.set("Bob", "London");
data.set("Alice", "Paris");
data.forEach((val, key) => console.log('${key} lives in ${val}'));
```

Коллекция `Map` была добавлена в JavaScript в рамках спецификации ES2015, и она не входит в версию, указанную в параметре `target` в файле `tsconfig.json`. При сохранении изменений в файле кода компилятор выдаст следующее предупреждение:

```
src/index.ts(6,16): error TS2583: Cannot find name 'Map'. Do you need to
change your target library? Try changing the 'lib' compiler option to
'es2015' or later.

6:50:49 AM - Found 1 error. Watching for file changes.
```

Чтобы решить эту проблему, можно перейти на более позднюю версию языка JavaScript или изменить определения типов, используемые компилятором, с помощью свойства конфигурации `lib`, которое устанавливается в массив значений из табл. 5.7.

Таблица 5.7. Значения для параметра компилятора `lib`

Название	Описание
<code>ES5, ES2015, ES2016, ES2017, ES2018, ES2019, ES2020, ES2021</code>	С помощью этих значений выбираются файлы определения типов, соответствующие конкретной версии спецификации JavaScript. При этом может использоваться и старая схема именования (например, вместо <code>ES2015</code> допустимо указывать <code>ES6</code>)
<code>ESnext</code>	Позволяет выбирать функции, которые предлагаются в качестве дополнений к спецификации JavaScript, но еще не были официально приняты. Набор функций будет меняться с течением времени
<code>DOM</code>	Данный параметр позволяет выбрать файлы с информацией о типах для API DOM, которые используются веб-приложениями для управления HTML-содержимым, отображаемым в браузерах. Этот параметр также полезен для приложений Node.js
<code>WebWorker</code>	Позволяет выбрать информацию о типе для Web Worker, с помощью которого веб-приложения выполняют фоновые задачи

Существуют также значения, позволяющие выбирать конкретные функции из разных версий спецификации языка. В табл. 5.8 описаны наиболее полезные настройки отдельных функций.

Таблица 5.8. Полезные значения для отдельных функций параметра компилятора `lib`

Название	Описание
<code>es2015.Core</code>	Включает информацию о типах для основных возможностей, появившихся в ES2015
<code>es2015.Collection</code>	Включает информацию о типах для коллекций <code>Map</code> и <code>Set</code> , описанных в главах 4 и 13
<code>es2015.Generator</code> <code>es2015.Iterable</code>	Включают информацию о типах для функций генератора и итератора, описанных в главах 4 и 13
<code>es2015.Promise</code>	Включает информацию о типах для промисов, которые описывают асинхронные действия
<code>es2015.Reflect</code>	Включает информацию о типах для функций отражения, обеспечивающих доступ к свойствам и прототипам, как описано в части III

Важно понимать последствия использования параметра конфигурации `lib`, поскольку он просто сообщает компилятору TypeScript, что среда выполнения приложения может рассчитывать на поддержку определенного набора функций, например `Map` в данном случае. Компилятор способен адаптировать генерируемый

им JavaScript для различных версий языка, но это не распространяется на такие объекты, как коллекции. Изменение параметра `lib` сообщает компилятору, что при выполнении скомпилированного JavaScript будет доступен нестандартный набор функций, и вы обязаны убедиться в этом, либо потому, что знаете о среде выполнения больше, чем компилятор, либо потому, что в приложении используется полифил, например `core-js` (<https://github.com/zloirock/core-js>).

Версия Node.js, установленная в главе 2, поддерживает большинство последних возможностей JavaScript, включая `Map`. Это значит, что можно безопасно изменять параметр `lib` в файле `tsconfig.json`, как показано в листинге 5.22.

Листинг 5.22. Изменение конфигурации в файле `tsconfig.json` из папки `tools`

```
{
  "compilerOptions": {
    "target": "ES5",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true,
    "lib": ["es5", "dom", "es2015.collection"]
  }
}
```

Выбранный набор типов включает стандартные типы для версии JavaScript, указанной в свойстве `target`, параметр `dom` (обеспечивающий доступ к объекту `console`) и функции коллекций ES2015 из табл. 5.8.

Компилятор обнаружит изменение в конфигурационном файле и перекомпилирует код. Изменение параметра `lib` сообщает компилятору, что `Map` будет доступна, и ошибка не выдается. При выполнении кода компилятор выведет следующий результат:

```
Message: Hello, TypeScript
Bob lives in London
Alice lives in Paris
```

Данный пример работает, поскольку версия Node.js, используемая в книге, поддерживает функцию `Map`. В данной ситуации разработчик знает о среде выполнения больше, чем компилятор TypeScript, и изменение параметра `lib` позволило примеру успешно сработать. Однако того же эффекта можно было бы добиться, изменив параметр `target` на более позднюю версию JavaScript, которая, как известно компилятору, включает коллекции. Если бы мы ориентировались на среду выполнения, поддерживающую только ES5, то нам пришлось бы предоставить полифильную реализацию `Map`, подобную той, что включена в пакет `core-js`.

5.8. ВЫБОР ФОРМАТА МОДУЛЯ

В главе 4 было описано, как с помощью модулей разбить JavaScript-приложение на несколько файлов для упрощения управления проектом. Компилятор TypeScript можно настроить на указание формата модуля, используемого в генерируемом им JavaScript, что гарантирует возможность выполнения выходного кода целевой средой выполнения.

В качестве демонстрации добавьте в папку `src` файл `calc.ts` с кодом из листинга 5.23.

Листинг 5.23. Содержимое файла `calc.ts` в папке `src`

```
export function sum(...vals: number[]): number {
    return vals.reduce((total, val) => total += val);
}
```

В этом файле используется ключевое слово `export` для создания функции с именем `sum`, которая вычисляет итоговую сумму для массива числовых значений. В листинге 5.24 эта функция импортируется в файл `index.ts` и вызывается из него.

Листинг 5.24. Использование модуля в файле `index.ts` из папки `src`

```
import { sum } from "./calc";

let printMessage = (msg: string): void =>
    console.log('Message: ${ msg }');

let message = ("Hello, TypeScript");
printMessage(message);

let total = sum(100, 200, 300);
console.log('Total: ${total}'');
```

После сохранения файла компилятор обработает код и выдаст следующий вывод:

```
Message: Hello, TypeScript
Total: 600
```

Изучив содержимое файла `index.js` в папке `dist`, вы увидите, что компилятор TypeScript ввел код для работы с модулями:

```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
var calc_1 = require("./calc");
var printMessage = function (msg) { return console.log("Message: ".concat(msg)); };
var message = ("Hello, TypeScript");
printMessage(message);
var total = (0, calc_1.sum)(100, 200, 300);
console.log("Total: ".concat(total));
```

Компилятор TypeScript использует свойство конфигурации `target` для выбора метода работы с модулями. Если установлено значение `ES5`, то используется стиль CommonJS модуля, который стал результатом более ранней попытки ввести стандарт модуля до того, как ECMAScript получил широкое распространение. Среда выполнения Node.js поддерживает систему модулей `commonJS`, поэтому код, сгенерированный компилятором TypeScript, выполняется без проблем.

При использовании более поздних версий языка JavaScript компилятор TypeScript переключается на формат модулей ECMAScript, а это означает, что ключевые слова `import` и `export` неизменными передаются из кода TypeScript в код JavaScript.

В листинге 5.25 изменена конфигурация компилятора для выбора версии JavaScript и удален параметр `lib`, чтобы компилятор использовал определения типов по умолчанию.

Листинг 5.25. Изменение конфигурации в файле tsconfig.json из папки tools

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true,
    // "lib": ["es5", "dom", "es2015.collection"]
  }
}
```

После сохранения изменений в конфигурационном файле компилятор заново генерирует JavaScript, но уже с использованием стандартных модулей. Однако Node.js не поддерживает ECMAScript без некоторых дополнительных настроек, и при выполнении JavaScript-кода компилятор выдает следующую ошибку:

```
...
import { sum } from "./calc";
^^^^^
SyntaxError: Cannot use import statement outside a module
...

```

Первая необходимая модификация — это настройка проекта в файле `package.json`, чтобы указать Node.js на необходимость применения ECMAScript, как показано в листинге 5.26.

Листинг 5.26. Указание формата модуля в файле package.json из папки tools

```
{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onSuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^6.0.0",
    "typescript": "^5.0.2"
  },
  "type": "module"
}
```

Свойство `type` может быть установлено в значение `module` для модулей ECMAScript или `commonjs` для модулей CommonJS. Необходимо также внести еще

одно изменение — включить расширение файла в имя файла в операторе `import`, как показано в листинге 5.27.

Листинг 5.27. Добавление расширения файла в файл `index.ts` из папки `src`

```
import { sum } from "./calc.js";

let printMessage = (msg: string): void => console.log('Message: ${ msg }');

let message = ("Hello, TypeScript");
printMessage(message);

let total = sum(100, 200, 300);

console.log('Total: ${total}');
```

Подобное требование кажется странным, поскольку оно предполагает указание расширения файла JavaScript, который создается компилятором, в файле TypeScript. Команда разработчиков TypeScript придерживается принципа отказа от переписывания путей в операторах импорта, а это означает, что компонент пути оператора `import`, задающий имя файла, должен быть написан для среды выполнения JavaScript, а не для компилятора TypeScript.

Учитывая то, насколько компилятор TypeScript переписывает код, это кажется мне странным и неудобным ущущением, но вряд ли это изменится, поэтому операторы `import` должны быть написаны с расширением `js`, а не `ts`.

РАСШИРЕНИЯ ФАЙЛОВ, СПЕЦИФИЧНЫЕ ДЛЯ ФОРМАТА МОДУЛЯ

Альтернативой использованию файла `package.json` для указания формата модуля является использование расширений файла. Расширение `mjs` обозначает модуль ECMAScript, а расширение `cjs` — модуль CommonJS. Компилятор TypeScript поддерживает расширения `mts` и `cts` для файлов TypeScript, которые создают файлы JavaScript с расширениями `mts` и `cts`. Если вы используете эту функцию, вам все равно потребуется включить расширение файла JavaScript в оператор `import`, чтобы задействовать функции, определенные в модуле. Признак расширения файла используется в главе 15 в примере, который требует соответствия формату модуля, используемому пакетом стороннего производителя.

5.8.1. Указание формата модуля

Система модулей может быть явно выбрана с помощью настройки `module` в файле `tsconfig.json` с помощью значений из табл. 5.9.

Для веб-приложений, особенно тех, что созданы с помощью таких фреймворков, как React или Angular, формат модулей будет определяться инструментарием фреймворка. Он может включать в себя сборщик, объединяющий все модули в один

JavaScript-файл при развертывании, или загрузчик модулей, отправляющий HTTP-запросы на веб-сервер для получения файлов JavaScript по мере их необходимости. Примеры использования компилятора TypeScript с этими фреймворками будут рассмотрены в части 3.

Таблица 5.9. Форматы модулей

Название	Описание
None	Отключает модули
CommonJS	Выбирает формат модуля CommonJS, который поддерживается Node.js
AMD	Выбирает определение асинхронного модуля (AMD), поддерживаемое загрузчиком модулей RequireJS
System	Выбирает формат модуля, поддерживаемый загрузчиком модулей SystemJS
UMD	Выбирает формат модуля Universal Module Definition (UMD)
ES2015, ES6	Выбирает формат модуля, указанный в спецификации ES2016
ES2020	Выбирает формат модуля, указанный в спецификации ES2020, предусматривающей динамическую загрузку модулей
ES2022	Выбирает формат модуля, указанный в спецификации ES2022, который поддерживает инициализацию модуля асинхронными данными
ESNext	Выбирает возможности модуля, предложенные для следующей версии языка JavaScript
Node16	Это значение предназначено для Node.js с использованием модулей ECMAScript или модулей CommonJS в зависимости от расширений файлов и конфигурации в package.json
NodeNext	Выбирает возможности модуля, предложенные для следующей версии Node.js

Наиболее полезной настройкой для проектов, ориентированных на среду выполнения Node.js, является `Node16`, как показано в листинге 5.28, которая обеспечивает настройку типа модуля, создаваемого компилятором TypeScript, с помощью свойства `type` в файле `package.json` или расширений `mts` и `cts`. Свойство модуля можно вообще не задавать, и поведение по умолчанию будет работать. Но использование параметра `Node16` гарантирует, что несоответствие между компилятором TypeScript и Node.js не возникнет. В редакторе кода, возможно, будет выведено предупреждение, которое можно проигнорировать, мы устраним его в следующем листинге.

Листинг 5.28. Выбор формата модуля в файле `tsconfig.json` из папки `tools`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
```

```

    "rootDir": "./src",
    "noEmitOnError": true,
    // "lib": ["es5", "dom", "es2015.collection"]
    "module": "Node16"
}
}
}

```

Если посмотреть на файл `index.js` в папке `dist`, то можно увидеть, что сгенерированный JavaScript-код работает с файлом `calc.js` следующим образом:

```

...
import { sum } from "./calc.js";
...

```

Оператор `import` в файле TypeScript — это тот же оператор, что используется для модулей ECMAScript, поэтому компилятор TypeScript включает его в файл JavaScript без изменений. Листинг 5.29 показывает изменение свойства `type` в файле `package.json` для указания формата модуля `CommonJS`.

Листинг 5.29. Изменение формата модуля в файле `package.json` из папки `tools`

```

{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onSuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^6.0.0",
    "typescript": "^5.0.2"
  },
  "type": "commonjs"
}

```

Когда файл `package.json` будет сохранен, компилятор TypeScript запустится. Если еще раз просмотреть файл `index.js`, то можно заметить, что изменение в файле `package.json` привело к смене формата модуля:

```

...
Object.defineProperty(exports, "__esModule", { value: true });

const calc_js_1 = require("./calc.js");
...

```

Компилятор сгенерировал утверждения, необходимые для формата CommonJS, и при выполнении код выдает следующий результат:

```

Message: Hello, TypeScript
Total: 600

```

РАЗРЕШЕНИЯ МОДУЛЕЙ

Компилятор TypeScript может применять разные подходы к разрешению зависимостей модулей, которые он выбирает в соответствии с используемым форматом модуля. Двумя наиболее часто используемыми режимами являются классический, который позволяет искать модули в локальном проекте, и Node, когда поиск модулей выполняется в папке `node_modules`. Настройки по умолчанию подходят для большинства проектов, но их можно переопределить с помощью параметров конфигурации `moduleResolution` в файле `tsconfig.json`, используя значение `classic` или `node`.

5.9. ПОЛЕЗНЫЕ ПАРАМЕТРЫ КОНФИГУРАЦИИ КОМПИЛЯТОРА

Компилятор TypeScript поддерживает большое количество параметров конфигурации. В начале каждой главы части 2 приведены таблицы, в которых перечислены параметры компилятора, используемые в примерах. Таблица 5.10 представляет собой сводный перечень параметров компилятора, используемых в книге, с их кратким описанием. Многие из них в данный момент могут показаться бессмысленными, но каждый будет детально рассмотрен в контексте использования, и к концу книги все они станут понятны.

СОВЕТ

Полный набор поддерживаемых параметров компилятора можно найти по ссылке: <https://www.typescriptlang.org/docs/handbook/compiler-options.html>.

Таблица 5.10. Параметры компилятора TypeScript, используемые в книге

Название	Описание
<code>allowJs</code>	Включает в процесс компиляции файлы JavaScript
<code>allowSyntheticDefaultImports</code>	Разрешает импорт из модулей, в которых не объявлен экспорт по умолчанию. Данная опция используется для повышения совместимости кода
<code>baseUrl</code>	Задает корень местоположения для разрешения зависимостей модуля
<code>checkJs</code>	Указывает компилятору провести проверку JavaScript-кода на наличие распространенных ошибок
<code>declaration</code>	Создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода
<code>downlevelIteration</code>	Включает поддержку итераторов при работе со старыми версиями JavaScript

Название	Описание
<code>emitDecoratorMetadata</code>	Разрешает компилятору генерировать метаданные о типах. Используется вместе с параметром <code>experimentalDecorators</code>
<code>esModuleInterop</code>	Добавляет вспомогательный код для импорта из модулей, которые не объявляют экспорт по умолчанию. Используется совместно с опцией <code>allowSyntheticDefaultImports</code>
<code>experimentalDecorators</code>	Включает поддержку декораторов
<code>forceConsistentCasingInFileNames</code>	Гарантирует, что имена в операторах импорта соответствуют регистру имен в операторах <code>import</code>
<code>importHelpers</code>	Определяет, будет ли к JavaScript добавляться вспомогательный код для уменьшения общего объема создаваемого кода
<code>isolatedModules</code>	Рассматривает каждый файл как отдельный модуль, что повышает совместимость с инструментом Babel
<code>jsx</code>	Определяет, как обрабатываются HTML-элементы в файлах JSX/TSX
<code>jsxFactory</code>	Задает имя фабричной функции, которая используется для замены HTML-элементов в файлах JSX/TSX
<code>lib</code>	Выбирает файлы деклараций типов, которые использует компилятор
<code>module</code>	Задает формат используемых модулей
<code>moduleResolution</code>	Задает стиль разрешения модуля, который следует использовать для разрешения зависимостей
<code>noEmit</code>	Запрещает компилятору генерировать JavaScript-код, в результате чего он проверяет код только на наличие ошибок
<code>noImplicitAny</code>	Предотвращает неявное использование типа <code>any</code> , который компилятор использует, когда не может определить более конкретный тип
<code>noImplicitReturns</code>	Требует, чтобы все пути в функции возвращали результат
<code>noUncheckedIndexedAccess</code>	Запрещает доступ к свойствам, доступ к которым осуществляется через индексную сигнатуру без проверки на <code>undefined</code>
<code>noUnusedParameters</code>	Выдает предупреждение, если функция определяет неиспользуемые параметры

Таблица 5.10 (продолжение)

Название	Описание
<code>outDir</code>	Задает каталог для размещения файлов JavaScript
<code>paths</code>	Задает местоположения, используемые для разрешения зависимостей модулей
<code>resolveJsonModule</code>	Позволяет импортировать JSON-файлы, как будто они являются модулями
<code>rootDir</code>	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
<code>skipLibCheck</code>	Ускоряет компиляцию, пропуская обычную проверку файлов деклараций
<code>sourceMap</code>	Определяет, генерировать ли компилятору карты исходников для отладки
<code>strict</code>	Позволяет более строго проверять код TypeScript
<code>strictNullChecks</code>	Не допускает присваивания <code>null</code> и <code>undefined</code> в качестве значений для других типов
<code>suppressExcessPropertyErrors</code>	Предотвращает выдачу ошибок для объектов, определяющих свойства не в указанной форме
<code>target</code>	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода
<code>typeRoots</code>	Задает корневое расположение для поиска файлов деклараций
<code>types</code>	Задает список файлов деклараций, включаемых в процесс компиляции

РЕЗЮМЕ

В этой главе был представлен компилятор TypeScript, который отвечает за преобразование кода TypeScript в чистый JavaScript. Были рассмотрены настройки компилятора, продемонстрированы различные способы его использования, объяснено, как изменить версию языка JavaScript, а также описаны способы разрешения модулей. В конце главы был приведен список параметров конфигураций, используемых в книге, которые, возможно, пока могут показаться не совсем понятными, но по мере изучения примеров станут более прозрачными.

- Проекты TypeScript организованы так, что код TypeScript, написанный разработчиком, хранится отдельно от кода JavaScript, выполняемого средой выполнения.
- Инструменты TypeScript добавляются в проект с помощью стандартного менеджера пакетов JavaScript, такого как NPM, или его аналогов.

- Компилятор TypeScript обрабатывает файлы TypeScript в проекте, генерируя чистый JavaScript.
- Файл `tsconfig.json` используется для настройки процесса генерации JavaScript-файлов компилятором.
- Компилятор TypeScript может работать в режиме наблюдения, когда файлы TypeScript компилируются при обнаружении изменений, но для автоматического выполнения сгенерированных JavaScript-файлов требуется пакет стороннего производителя, например `tsc-watch`.
- Компилятор TypeScript может генерировать код, соответствующий различным версиям спецификации языка JavaScript.
- Компилятор TypeScript способен генерировать код, использующий различные форматы модулей JavaScript, и поддерживает те же параметры конфигурации, что и Node.js, для определения используемого формата.

В следующей главе будет продолжена тема инструментов разработчика TypeScript, включая выполнение отладки и модульного тестирования TypeScript-кода.

Тестирование и отладка TypeScript

В этой главе

- ✓ Отладка TypeScript-кода с помощью Visual Studio Code и отладчика Node.js.
- ✓ Использование linter для поиска проблем в коде, которые не обнаружил компилятор.
- ✓ Написание и выполнение модульных тестов для TypeScript-кода.

Здесь мы продолжим тему инструментов разработки TypeScript, начатую в главе 5, где был представлен компилятор TypeScript. Будут показаны различные методы отладки TypeScript-кода, а также объяснено, как использовать TypeScript и linter и как настроить модульное тестирование для TypeScript-кода.

6.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В этой главе все еще будем использовать проект `tools`, созданный в главе 5. Никаких изменений не потребуется.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Откройте новое окно командной строки, зайдите в папку `tools` и выполните команду, показанную в листинге 6.1, для запуска компилятора в режиме наблюдения с помощью пакета `tsc-watch`, установленного в главе 5.

Листинг 6.1. Запуск компилятора

```
npm start
```

Компилятор запустится, файлы TypeScript в проекте скомпилируются, и на экран будет выведен следующий результат:

```
7:04:50 AM - Starting compilation in watch mode...
7:04:52 AM - Found 0 errors. Watching for file changes.
Message: Hello, TypeScript
Total: 600
```

6.2. ОТЛАДКА TYPESCRIPT-КОДА

Компилятор TypeScript хорошо справляется с выявлением синтаксических ошибок и проблем с типами данных. Однако бывают случаи, когда код компилируется успешно, но при выполнении ведет себя не так, как ожидалось. Использование отладчика позволяет проследить за состоянием приложения в процессе его выполнения и выявить причины возможных проблем. В последующих разделах мы рассмотрим, как отлаживать приложение на TypeScript, запускаемое в Node.js. В части III вы увидите, как отлаживать веб-приложения на TypeScript.

6.2.1. Подготовка к отладке

Сложность отладки TypeScript-приложения заключается в том, что исполняемый код является результатом компиляции. Чтобы помочь отладчику сопоставить JavaScript-код с кодом TypeScript, компилятор может генерировать файлы, называемые *картами исходного кода*. В листинге 6.2 карты исходного кода включены в файл `tsconfig.json`.

Листинг 6.2. Включение карт исходного кода в файл `tsconfig.json` из папки `tools`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true,
    "module": "Node16",
    "sourceMap": true
  }
}
```

Когда компилятор в следующий раз скомпилирует файлы TypeScript, то наряду с файлами JavaScript в папке `dist` он сгенерирует и файл с расширением `map`.

Добавление точек останова

Редакторы кода с хорошей поддержкой TypeScript, такие как Visual Studio Code, позволяют добавлять точки останова в файлы кода. Однако мой опыт использования данной функции был неоднозначным, поэтому я полагаюсь на менее элегантное, но более предсказуемое выражение `debugger` в JavaScript. Когда JavaScript-приложение запускается через отладчик, выполнение останавливается при обнаружении выражения `debugger` и управление передается разработчику. Преимуществом такого подхода является его надежность и универсальность, однако перед развертыванием необходимо не забыть удалить выражение `debugger`. Большинство сред игнорируют `debugger` при обычном выполнении, но лучше не полагаться на подобное поведение. Линтер, описанный далее в этой главе, может помочь избежать оставления ключевого слова отладчика в файлах кода.

Добавьте выражение `debugger` в файл `index.ts`, как показано в листинге 6.3.

Листинг 6.3. Добавление выражения `debugger` в файл `index.ts` из папки `src`

```
import { sum } from "./calc.js";

let printMessage = (msg: string): void => console.log('Message: ${ msg }');

let message = ("Hello, TypeScript");
printMessage(message);

debugger;

let total = sum(100, 200, 300);
console.log('Total: ${total}');
```

При выполнении кода никаких изменений в выводе не произойдет, поскольку Node.js по умолчанию игнорирует ключевое слово `debugger`.

6.2.2. Использование Visual Studio Code для отладки

Большинство современных редакторов кода в той или иной степени поддерживают отладку TypeScript и JavaScript-кода. В этом разделе будет показано, как выполнять отладку с помощью Visual Studio Code, чтобы дать вам представление об этом процессе. Если вы используете другой редактор, могут потребоваться другие шаги, но основной подход, скорее всего, будет аналогичным.

Чтобы настроить конфигурацию для отладки в Visual Studio Code, выберите в меню Run пункт `Add Configuration` и при появлении запроса выберите `Node.js` из списка окружений, как показано на рис. 6.1.

ПРИМЕЧАНИЕ

Если это не сработает, попробуйте выбрать `Start Debugging` вместо `Add Configuration`.

Редактор автоматически создаст в проекте папку `.vscode` и добавит в нее файл с именем `launch.json`, который используется для настройки отладчика. Измените

значение свойства `program`, как показано в листинге 6.4, чтобы отладчик выполнял JavaScript-код из папки `dist`.

Листинг 6.4. Изменение пути к коду в файле `launch.json` из папки `.vscode`

```
{  
    "version": "0.2.0",  
    "configurations": [  
        {  
            "type": "node",  
            "request": "launch",  
            "name": "Launch Program",  
            "skipFiles": [  
                "<node_internals>/**"  
            ],  
            "program": "${workspaceFolder}/dist/index.js",  
            "outFiles": [  
                "${workspaceFolder}/**/*.js"  
            ]  
        }  
    ]  
}
```

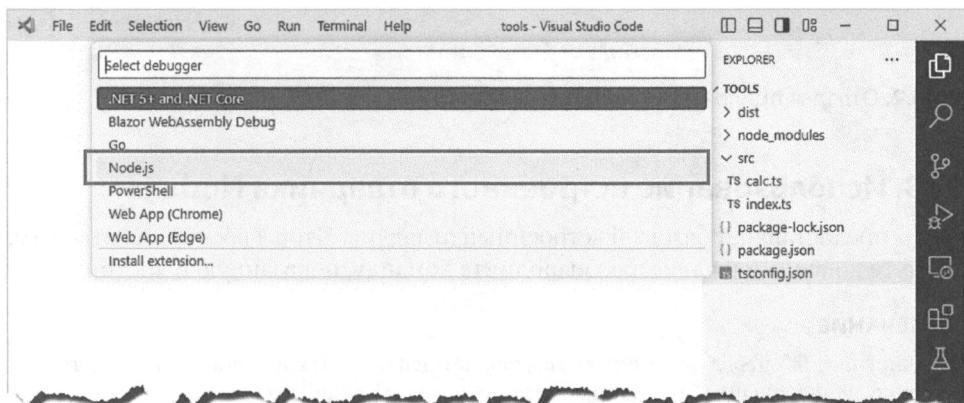


Рис. 6.1. Выбор среды отладчика

Сохраните изменения в файле `launch.json` и выберите в меню `Run` пункт `Start Debugging`. Visual Studio Code выполнит файл `index.js` в папке `dist` под управлением отладчика `Node.js`. Выполнение будет продолжаться в обычном режиме до тех пор, пока не будет достигнуто выражение `debugger`, после чего выполнение прекратится и откроется всплывающее окно отладки, как показано на рис. 6.2.

Состояние приложения отображается на боковой панели, показывая переменные, которые были установлены на момент остановки выполнения. Вам доступен стандартный набор функций отладки, включая установку точек останова, переход к операторам и возобновление выполнения. Okno `Debug Console` позволяет выполнять операторы `JavaScript` в контексте приложения. Например,

при вводе имени переменной и нажатии Enter возвращается значение, присвоенное этой переменной.

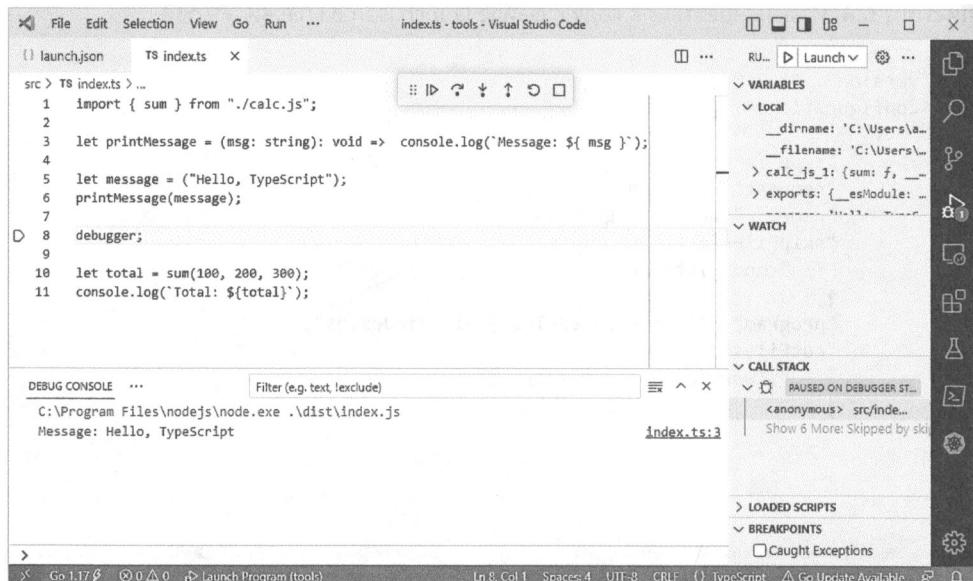


Рис. 6.2. Отладка приложения с помощью Visual Studio Code

6.2.3. Использование встроенного отладчика Node.js

Node.js предоставляет базовый встроенный отладчик. Чтобы воспользоваться им, откройте консоль и в папке tools выполните команду, показанную в листинге 6.5.

ПРИМЕЧАНИЕ

В листинге 6.5 перед аргументом `inspect` нет дефисов. Их наличие означает, что мы используем удаленный отладчик, описанный в следующем разделе.

Листинг 6.5. Запуск отладчика Node.js

```
node inspect dist/index.js
```

Отладчик запускается, загружает файл `index.js` и останавливает выполнение. Чтобы продолжить, введите команду, показанную в листинге 6.6, и нажмите Enter, чтобы продолжить выполнение.

Листинг 6.6. Продолжение выполнения

```
c
```

При достижении выражения `debugger` отладчик снова останавливается. С помощью команды `exec` можно выполнять выражения для проверки состояния приложений, при этом выражения должны быть заключены в кавычки как строки. Введите в окне отладчика команду, показанную в листинге 6.7.

Листинг 6.7. Вычисление выражения в отладчике Node.js

```
exec("message")
```

Нажмите Enter, и отладчик отобразит значение переменной `message`, выдав следующий результат:

```
'Hello, TypeScript'
```

Если вам нужна справка по командам, введите `help` и нажмите Enter. Для завершения сеанса отладки и возврата к обычному командному интерпретатору дважды нажмите `Control+C`.

6.2.4. Функция удаленной отладки Node.js

Встроенный отладчик Node.js полезен, но не очень удобен в использовании. Те же возможности можно использовать удаленно с помощью инструментария разработчика Google Chrome. Для начала запустите Node.js в папке `tools`, выполнив команду, показанную в листинге 6.8.

Листинг 6.8. Запуск Node.js в режиме удаленного отладчика

```
node --inspect-brk dist/index.js
```

Аргумент `inspectBrk` запускает отладчик и немедленно останавливает выполнение. Это необходимо для примера приложения, поскольку оно запускается, а затем завершается. Для приложений, которые запускаются и затем переходят в бесконечный цикл, например для веб-сервера, можно использовать аргумент `inspect`. При запуске Node.js выдаст сообщение, подобное этому:

```
Debugger listening on ws://127.0.0.1:9229/e3cf5393-23c8-4393-99a1  
For help, see: https://nodejs.org/en/docs/inspector
```

URL-адрес в выводе используется для подключения к отладчику и управления выполнением. Откройте новое окно Chrome и перейдите по адресу `chrome://inspect`. Нажмите кнопку `Configure` и добавьте IP-адрес и порт из URL-адреса из предыдущего сообщения. Для моего компьютера это `127.0.0.1:9229` (рис. 6.3).

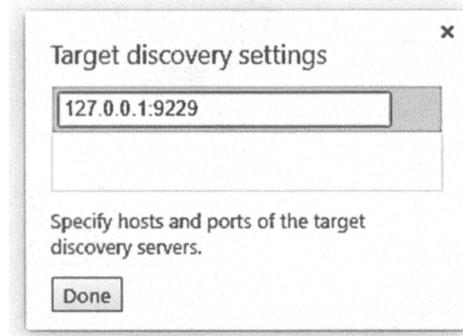


Рис. 6.3. Настройка Chrome для удаленной отладки Node.js

Нажмите кнопку **Done** и подождите некоторое время, пока Chrome найдет среду выполнения Node.js. После того как она будет найдена, она появится в списке **Remote Target** (Удаленный объект), как показано на рис. 6.4.

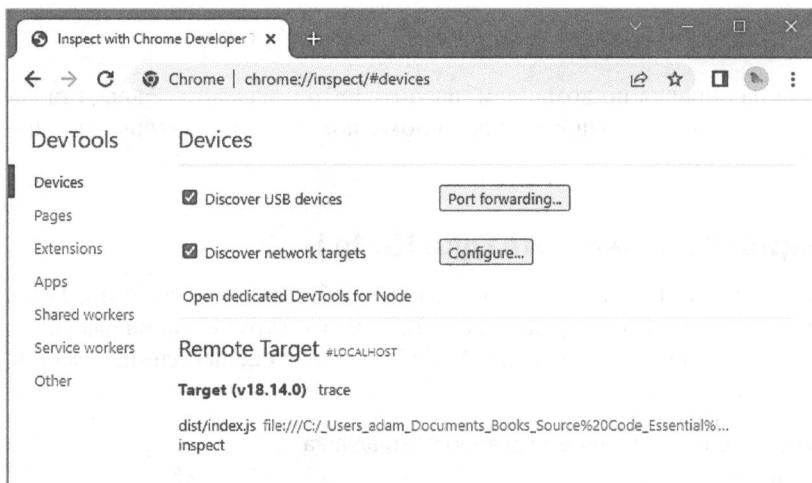


Рис. 6.4. Обнаружение среды выполнения Node.js

Щелкните на ссылке **inspect**, чтобы открыть новое окно инструментов разработчика Chrome, подключенное к среде выполнения Node.js. Управление выполнением осуществляется с помощью стандартных кнопок инструментов разработчика, а возобновление выполнения позволит среде выполнения продолжить работу до тех пор, пока не будет достигнуто выражение **debugger**. Первоначально в окне отладчика отображается JavaScript-код, но после возобновления выполнения будут использоваться карты исходного кода, как показано на рис. 6.5.

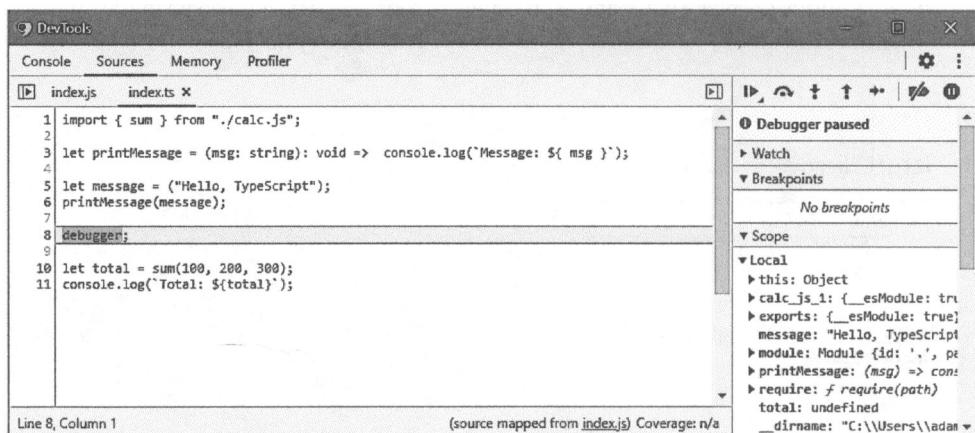


Рис. 6.5. Отладка с помощью инструментов разработчика Chrome

6.3. ЛИНТЕР TYPESCRIPT

Линтер – это инструмент, анализирующий файлы кода с использованием набора правил, описывающих проблемы, которые могут вызывать путаницу, привести к неожиданным результатам или ухудшить читаемость кода. Стандартным линтером для TypeScript является `typescript-eslint`, который адаптирует популярный JavaScript-линтер `eslint` для работы с TypeScript. Чтобы добавить линтер в проект, выполните команды, показанные в листинге 6.9, находясь в папке `tools`.

ПРИМЕЧАНИЕ

Ранее стандартным линтером для TypeScript был TSLint, но `typescript-eslint` оказался конкурентоспособнее.

Листинг 6.9. Добавление пакетов в проект примера

```
npm install --save-dev eslint@8.36.0
npm install --save-dev @typescript-eslint/parser@5.55.0
npm install --save-dev @typescript-eslint/eslint-plugin@5.55.0
```

Чтобы настроить конфигурацию, необходимую для использования линтера, добавьте в папку `tools` файл с именем `.eslintrc`, содержимое которого показано в листинге 6.10.

Листинг 6.10. Содержимое файла `.eslintrc` в папке `tools`

```
{
  "root": true,
  "ignorePatterns": ["node_modules", "dist"],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json"
  },
  "plugins": [
    "@typescript-eslint"
  ],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/eslint-recommended",
    "plugin:@typescript-eslint/recommended"
  ]
}
```

Линтер поставляется с предварительно настроенными наборами правил, которые задаются с помощью параметра `extends`, как описано в табл. 6.1.

Остановите процесс `node` с помощью `Control+C`, войдите в папку `tools` и запустите линтер командой из листинга 6.11 (не пропустите точку в конце команды).

Листинг 6.11. Запуск линтера TypeScript

```
npx eslint .
```

Таблица 6.1. Предварительно настроенные наборы правил линтера

Название	Описание
eslint:recommended	Набор правил, рекомендованных командой разработчиков ESLint для общей разработки на JavaScript
@typescript-eslint/eslint-recommended	Данный набор переопределяет набор recommended для отключения правил, которые не требуются для проверки кода TypeScript
@typescript-eslint/recommended	Набор содержит дополнительные правила, рекомендованные для TypeScript

Аргумент `project` указывает линтеру на использование файла настроек компилятора для поиска проверяемых исходных файлов. В нашем примере проекта имеется только один файл TypeScript. Линтер проанализирует код и выдаст следующий результат:

```
C:\tools\src\index.ts
```

```
3:5 error 'printMessage' is never reassigned.
  Use 'const' instead prefer-const

5:5 error 'message' is never reassigned.
  Use 'const' instead      prefer-const
8:1 error Unexpected 'debugger' statement no-debugger
10:5 error 'total' is never reassigned.
  Use 'const' instead      prefer-const

4 problems (4 errors, 0 warnings)
3 errors and 0 warnings potentially fixable with the '--fix' option.
```

Линтер находит файлы TypeScript-кода и проверяет их на соответствие правилам, заданным в конфигурационном файле. Код в примере проекта имеет два нарушения правил линтера: правило `prefer-const` требует использовать ключевое слово `const` вместо `let`, если значение переменной не изменяется, а правило `no-debugger` запрещает использовать выражение `debugger`.

6.3.1. Отключение правил линтера

Однако стоит помнить, что ценность того или иного правила линтинга часто зависит от индивидуального стиля и предпочтений разработчика. Более того, даже если правило полезно, оно не всегда применимо в каждой ситуации. Линтер работает наиболее эффективно, когда программист видит только те предупреждения, которые действительно требуют внимания. Если вы получили список предупреждений, которые вас не волнуют, существует риск пропустить среди прочих важные сообщения.

Например, правило `prefer-const` подчеркивает недостаток моего стиля кодирования, но я знаю, что мне следует использовать `const` вместо `let`, и стараюсь

придерживаться этого принципа. Однако мои привычки в кодировании глубоко укоренились, и я считаю, что не все «проблемы» стоит исправлять, особенно если это только отвлекает от работы. Я признаю свои недостатки и понимаю, что продолжу использовать `let`, даже если `const` в определенный момент был бы предпочтительнее. Я не хочу, чтобы линтер подчеркивал данный аспект, и поэтому соответствующе настроил его на отключение правил, как показано в листинге 6.12.

Листинг 6.12. Отключение правил линтера в файле `.eslintrc` из папки `tools`

```
{
  "root": true,
  "ignorePatterns": ["node_modules", "dist"],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json"
  },
  "plugins": [
    "@typescript-eslint"
  ],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/eslint-recommended",
    "plugin:@typescript-eslint/recommended"
  ],
  "rules": {
    "prefer-const": 0
  }
}
```

Раздел конфигурации `rules` заполняется именами правил и значениями для их включения (1) или отключения (0). Установив значение 0 для правила `prefer-const`, я дал команду линтеру игнорировать использование ключевого слова `let`, если `const` в этот момент является более предпочтительным.

Некоторые правила могут быть полезны в рамках проекта, но не всегда подходят для конкретных файлов или операторов. Именно к такой категории относится правило `no-debugger`. Обычно ключевое слово `debugger` не следует оставлять в файлах кода, поскольку это может привести к проблемам во время выполнения кода. Однако при исследовании проблемы отладчик является полезным способом надежного управления выполнением приложения, как было продемонстрировано ранее в этой главе. В таких ситуациях нет смысла отключать правило в конфигурационном файле линтера. Вместо этого лучше использовать комментарий, начинающийся с `eslint-disable-line`, за которым следуют одно или несколько имен правил, чтобы отключить их для конкретного оператора, как показано в листинге 6.13.

Листинг 6.13. Отключение правила для одного оператора в файле `index.ts` из папки `src`

```
import { sum } from "./calc.js";

let printMessage = (msg: string): void => console.log('Message: ${ msg }');
```

```
let message = ("Hello, TypeScript");
printMessage(message);

debugger; // eslint-disable-line no-debugger

let total = sum(100, 200, 300);
console.log('Total: ${total}');
```

Комментарий в листинге 6.13 указывает линтеру не применять правило `no-debugger` к выделенному утверждению. Выполните команду из листинга 6.11 еще раз, и вы увидите, что изменение конфигурации и комментарий для линтера подавляют предыдущие предупреждения.

СОВЕТ

Правила проверки могут быть отключены для всех утверждений, следующих за блочным комментарием (начинающимся с `/*` и заканчивающимся `*/`), который начинается с `eslint-disable`. Это позволяет отключить все правила проверки, используя комментарий `eslint-disable` или `eslint-disable-line` без указания имен конкретных правил.

РАДОСТИ И ГОРЕСТИ ЛИНТИНГА

Линтеры могут стать мощным инструментом, особенно в команде разработчиков с разным уровнем квалификации и опыта. Линтеры позволяют обнаружить распространенные проблемы и тонкие ошибки, которые приводят к неожиданному поведению или проблемам сопровождения кода в долгосрочной перспективе. Мне нравится этот вид проверки, и я предпочитаю прогонять свой код через линтеры после завершения работы над основной функциональностью приложения или перед фиксацией изменений кода в системе контроля версий.

Однако линтеры способны стать источником споров и разногласий. Помимо выявления ошибок в коде, линтеры могут использовать для обеспечения соблюдения правил форматирования, таких как применение отступов, расстановка фигурных скобок, использование точек с запятой и пробелов, и множества других аспектов стиля кодирования. Каждый разработчик имеет свои стилевые предпочтения, которых он придерживается, и часто считает, что остальные тоже должны этому следовать.

Я, к слову, предпочитаю использовать четыре пробела для отступов и располагать открывающие скобки на одной строке с соответствующим выражением. Это часть моего «единственно верного способа» написания кода, и тот факт, что другие программисты предпочитают, например, два пробела, вызывает у меня тихое изумление.

Линтеры позволяют людям, имеющим твердые взгляды на форматирование, навязывать их другим, обычно прикрываясь тем, что у каждого «есть свое мнение». Логика тут проста — чтобы избежать трат времени на бесконечные

споры о стиле кодирования, лучше заставить всех писать одинаково. Однако мой опыт показывает, что разработчики все равно найдут о чем поспорить и принудительное введение единого стиля кода часто приводит к тому, что предпочтения одного навязываются всей команде.

Я часто помогаю читателям, сталкивающимся с трудностями при работе с примерами из книги (мой адрес электронной почты: adam@adamfreeman.com), и постоянно вижу самые разные стили кодирования. В глубине души я знаю, что любой, кто не следует моим личным предпочтениям в кодировании, просто ошибается. Поэтому вместо того, чтобы заставлять кого-либо писать по-моему, я переформатирую код с помощью возможностей своего редактора кода — эту функцию предоставляет каждый продвинутый редактор.

Мой совет — использовать линтеры в редких случаях и сосредотачиваться на действительно важных для проекта проблемах. Оставьте свободу выбора форматирования разработчикам и полагайтесь на переформатирование редактора, если вам нужно прочитать код, написанный членом команды с другими предпочтениями.

6.4. МОДУЛЬНОЕ ТЕСТИРОВАНИЕ TYPESCRIPT

Некоторые фреймворки для модульного тестирования обеспечивают поддержку TypeScript, хотя это не так полезно, как может показаться. Поддержка TypeScript в модульном тестировании позволяет определять тесты в файлах TypeScript и иногда автоматически компилировать их перед запуском теста. Модульные тесты выполняются путем выполнения небольших частей приложения, и это можно сделать только с помощью JavaScript, поскольку среды выполнения JavaScript не имеют представления о возможностях TypeScript. В результате модульное тестирование не может быть использовано для тестирования функций TypeScript, которые реализуются исключительно компилятором TypeScript.

При работе над книгой я применял тестовый фреймворк Jest, который прост в использовании и поддерживает тесты на TypeScript. Кроме того, при добавлении дополнительного пакета он обеспечит компиляцию файлов TypeScript в JavaScript перед выполнением тестов. Выполните команды, показанные в листинге 6.14, в папке tools, чтобы установить пакеты, необходимые для тестирования.

Листинг 6.14. Добавление пакетов в проект

```
npm install --save-dev jest@29.4.3
npm install --save-dev ts-jest@29.0.5
npm install --save-dev @types/jest@29.4.0
```

Пакет `jest` содержит фреймворк тестирования. Пакет `ts-jest` — это плагин к Jest, он отвечает за компиляцию файлов TypeScript перед запуском тестов. Пакет `@types/jest` содержит определения TypeScript для API Jest.

РЕШЕНИЕ О ПРОВЕДЕНИИ МОДУЛЬНОГО ТЕСТИРОВАНИЯ

Модульное тестирование — это спорная тема. В данном разделе предполагается, что вы уже решили заняться модульным тестированием. Здесь будет показано, как настроить инструменты и применить их в TypeScript. Это не введение в модульное тестирование, и я не пытаюсь переубедить скептически настроенных читателей в целесообразности его проведения. Если вам нужна более подробная информация, рекомендую прочитать хорошую статью: https://en.wikipedia.org/wiki/Unit_testing.

Мне нравится модульное тестирование, и я использую его в своих проектах, но не во всех и не так последовательно, как можно было бы ожидать. Как правило, я фокусируюсь на написании модульных тестов для тех функций и возможностей, которые, как мне кажется, будут наиболее сложными и, скорее всего, станут источником ошибок при развертывании. В таких ситуациях модульное тестирование помогает структурировать мои мысли о том, как наилучшим образом реализовать необходимую функциональность. Я обнаружил, что простое размышление о том, что следует протестировать, помогает выявить потенциальные проблемы еще до того, как я начну разбираться с реальными ошибками и дефектами.

Однако стоит помнить, что модульное тестирование — это инструмент, а не догма, и только вы знаете, какой объем тестирования необходим в вашем проекте. Если вы считаете, что модульное тестирование не принесет пользы или у вас есть другая методология, которая вам ближе, не стоит проводить модульное тестирование только потому, что это модно (в то же время если у вас нет альтернативы и вы вообще не проводите тесты, то, скорее всего, вы оставляете возможность пользователям находить ваши ошибки, что редко бывает идеальным решением).

6.4.1. Настройка тестового фреймворка

Для настройки Jest добавьте в папку `tools` файл с именем `jest.config.js`, содержимое которого показано в листинге 6.15.

Листинг 6.15. Содержимое файла `jest.config.js` в папке `tools`

```
module.exports = {
  "roots": ["src"],
  "transform": {"^.+\\.tsx?$": "ts-jest"}
}
```

Параметр `roots` используется для указания местоположения файлов кода и модульных тестов. Свойство `transform` сообщает Jest, что файлы с расширением `ts`

и `tsx` следует обрабатывать с помощью пакета `ts-jest`. Это позволяет автоматически отслеживать изменения в коде при выполнении тестов, избегая необходимости явного запуска компилятора. (Файлы TSX описаны в главе 16.)

6.4.2. Создание модульных тестов

Тесты определяются в файлах с расширением `test.ts` и, как правило, создаются вместе с файлами кода, к которым они относятся. Чтобы создать простой модульный тест для примера приложения, добавьте в папку `src` файл `calc.test.ts` и поместите в него код, показанный в листинге 6.16.

Листинг 6.16. Содержимое файла `calc.test.ts` из папки `src`

```
import { sum } from "./calc";

test("check result value", () => {
  let result = sum(10, 20, 30);
  expect(result).toBe(60);
});
```

Тесты определяются с помощью функции `test`, предоставляемой Jest. Аргументами `test` являются имя теста и функция, выполняющая тестирование. Модульный тест в листинге 6.16 называется `check result value`. Он вызывает функцию `sum` с тремя аргументами и проверяет результат. Jest предоставляет функцию `expect`, которая передает результат и используется с функцией сопоставления (matcher function) для определения ожидаемого результата. В листинге 6.16 для сопоставления используется функция `toBe`, которая сообщает Jest, что ожидается конкретное значение. В табл. 6.2 описаны наиболее полезные функции сопоставления (полный их список можно найти на сайте <https://jestjs.io/docs/en/expect>).

Таблица 6.2. Полезные функции сопоставления Jest

Название	Описание
<code>toBe(value)</code>	Утверждает, что результат совпадает с заданным значением (но не обязательно с тем же самым объектом)
<code>toEqual(object)</code>	Утверждает, что результат является тем же объектом, что и указанное значение
<code>toMatch(regexp)</code>	Утверждает, что результат соответствует заданному регулярному выражению
<code>toBeDefined()</code>	Утверждает, что результат определен
<code>toBeUndefined()</code>	Утверждает, что результат не был определен
<code>toBeNull()</code>	Утверждает, что результат равен null
<code>toBeTruthy()</code>	Утверждает, что результат является истинным

Таблица 6.2 (продолжение)

Название	Описание
<code>toBeFalsy()</code>	Утверждает, что результат является ложным
<code>toContain(substring)</code>	Утверждает, что результат содержит указанную подстроку
<code>toBeLessThan(value)</code>	Утверждает, что результат меньше заданного значения
<code>toBeGreaterThan(value)</code>	Утверждает, что результат больше заданного значения

Обратите внимание, что в операторе `import` из листинга 6.16 не указано расширение файла. Это связано с тем, что при публикации пакета Jest используется формат модулей CommonJS, а не ECMAScript, на который настроен TypeScript. Как отмечалось ранее, пройдет некоторое время, прежде чем все перейдут на модули ECMAScript, а до тех пор следует внимательно относиться к именам файлов в операторах `import`.

6.4.3. Запуск тестового фреймворка

Модульные тесты можно запускать как одноразовую задачу или использовать режим наблюдения, который запускает тесты при обнаружении изменений. Для меня наиболее удобен режим наблюдения, поэтому у меня открыты две командные строки: одна для вывода компилятора, другая — для модульных тестов. Чтобы запустить тесты, откройте консоль, перейдите в папку `tools` и выполните команду, показанную в листинге 6.17. Предупреждения о несоответствии версий, выдаваемые пакетом `ts-jest`, можно игнорировать.

Листинг 6.17. Запуск фреймворка модульного тестирования в режиме наблюдения
`npx jest --watchAll`

Jest запустится, найдет в проекте тестовые файлы и выполнит их, выдав следующий результат:

```
PASS src/calc.test.ts
  check result value (3ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.214s
Ran all test suites.
Watch Usage
  ' Press f to run only failed tests.
  ' Press o to only run tests related to changed files.
  ' Press p to filter by a filename regex pattern.
  ' Press t to filter by a test name regex pattern.
  ' Press q to quit watch mode.
  ' Press Enter to trigger a test run.
```

Вывод показывает, что Jest обнаружил один тест и успешно его выполнил. Если будут определены дополнительные тесты или изменен исходный код приложения, Jest выполнит тесты снова и выдаст новый отчет. Чтобы увидеть, что происходит

при неудачном выполнении теста, внесите изменения, показанные в листинге 6.18, в функцию `sum`, которая является объектом тестирования.

Листинг 6.18. Умышленное внесение ошибки в тест в файле calc.ts из папки src

```
export function sum(...vals: number[]): number {
    return vals.reduce((total, val) => total += val) + 10;
}
```

Функция `sum` больше не возвращает ожидаемое тестом значение, и Jest выдает следующее предупреждение:

```
FAIL src/calc.test.ts
  check result value (6ms)
    check result value
      expect(received).toBe(expected) // Object.is equality
        Expected: 60
        Received: 70
          3 |   test("check result value", () => {
          4 |     let result = sum(10, 20, 30);
    > 5 |     expect(result).toBe(60);
          |
          6 |   });
      at Object.<anonymous> (src/calc.test.ts:5:20)
Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:  0 total
Time:        4.726s
Ran all test suites.
Watch Usage: Press w to show more.
```

Выходные данные показывают результат, ожидаемый тестом, и результат, который был получен. В случае неудачного теста можно принять одно из двух решений: либо поправить исходный код в соответствии с ожиданиями теста, либо, если требования к исходному коду изменились, обновить сам тест, чтобы он отражал новое поведение. В листинге 6.19 изменен модульный тест.

Листинг 6.19. Изменение юнит-теста в файле calc.test.ts в папке src

```
import { sum } from "./calc";

test("check result value", () => {
  let result = sum(10, 20, 30);
  expect(result).toBe(70);
});
```

После сохранения изменений в teste Jest снова запускает тесты и сообщает об их успешном выполнении.

```
PASS src/calc.test.ts
  check result value (3ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:  0 total
Time:        5s
Ran all test suites.
Watch Usage: Press w to show more.
```

РЕЗЮМЕ

В этой главе мы рассмотрели три инструмента, которые часто применяются для поддержки разработки на TypeScript. Отладчик Node.js позволяет проверить состояние приложений в процессе их выполнения, линтер помогает избежать распространенных ошибок в коде, которые не выявляются компилятором, но способны вызвать проблемы, а фреймворк модульных тестов используется для подтверждения того, что код ведет себя так, как ожидается. Отладка TypeScript может осуществляться с помощью интегрированного отладчика, входящего в состав Visual Studio Code, или с помощью отладчика, встроенного в Node.js.

- Точки останова можно создавать с помощью редактора кода или с помощью ключевого слова `debugger`.
- Линтер TypeScript проверяет код TypeScript на наличие распространенных проблем.
- TypeScript использует сторонние фреймворки, такие как Jest, для модульного тестирования.

В следующей главе мы подробнее рассмотрим возможности TypeScript, начиная со статической проверки типов.

Часть II

Возможности TypeScript



Статические типы

В этой главе

- ✓ Ограничение диапазона типов для переменных и операций.
- ✓ Роль компилятора при выводе типов.
- ✓ Использование типов `any`, `never` и `unknown` для расширения или ограничения диапазона значений.
- ✓ Объединения типов.
- ✓ Использование утверждений типов и средств защиты типа.
- ✓ Контроль использования значений `null` и `undefined` в JavaScript.

В этой главе мы рассмотрим ключевые возможности TypeScript для работы с типами данных. Функции, описанные здесь, являются фундаментом TypeScript и служат строительными блоками для более сложных функций, описанных в последующих главах.

Сначала демонстрируются различия между типами в TypeScript и чистом JavaScript. Затем вы узнаете, что компилятор TypeScript может выводить типы данных из кода, а также познакомитесь с возможностями, позволяющими более точно контролировать типы данных либо путем предоставления компилятору TypeScript дополнительной информации о том, как должны вести себя участки кода, либо путем изменения настройки компилятора.

В табл. 7.1 приведено краткое содержание главы.

В качестве краткой справки в табл. 7.2 перечислены параметры компилятора TypeScript, используемые в этой главе.

Таблица 7.1. Краткое содержание главы

Задача	Решение	Листинг
Указать тип	Используйте аннотацию типа или разрешите компилятору вывести тип	10–13
Просмотреть типы, которые компилятор выводит	Включите опцию компилятора <code>declarations</code> и просмотрите скомпилированный код	14, 15
Разрешить использование любого типа	Укажите тип <code>any</code> или <code>unknown</code>	16–19, 29, 30
Запретить компилятору выводить тип <code>any</code>	Включите опцию компилятора <code>noImplicitAny</code>	20
Объединить типы	Используйте объединения типов	21, 22
Переопределить тип, ожидаемый компилятором	Используйте утверждения типа	23–25
Проверить на примитивный тип значения	Используйте оператора <code>typeof</code> в качестве защиты типа (type guard)	26–28
Предотвратить появление <code>null</code> или <code>undefined</code> в значениях других типов	Включите опцию компилятора <code>strictNullChecks</code>	31–33
Отменить в компиляторе исключение значения <code>null</code> из объединения	Используйте утверждение <code>non-null</code> или защиту типа (type guard)	34, 35
Разрешить использовать переменную, если у нее нет значения	Используйте утверждение определенного присваивания	36, 37

Таблица 7.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
<code>declaration</code>	Создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода. Более подробно эти файлы описаны в главе 15
<code>noImplicitAny</code>	Предотвращает неявное использование типа <code>any</code> , который компилятор использует, когда не может определить более конкретный тип
<code>outDir</code>	Задает каталог, в котором будут размещены файлы JavaScript
<code>rootDir</code>	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
<code>strictNullChecks</code>	Не допускает присваивание <code>null</code> и <code>undefined</code> в качестве значений для других типов
<code>target</code>	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

7.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

Для работы с примером проекта для этой главы создайте в удобном для вас месте папку `types`. Откройте консоль, перейдите в каталог `types` и выполните команду, показанную в листинге 7.1, чтобы инициализировать папку для использования с NPM.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 7.1. Инициализация менеджера пакетов Node

```
npm init --yes
```

Добавьте необходимые пакеты, выполнив в папке `types` команду из листинга 7.2.

Листинг 7.2. Добавление пакетов в проект

```
npm install --save-dev typescript@5.0.2
npm install --save-dev tsc-watch@6.0.0
```

Настройте компилятор TypeScript, поместив в папку `types` файл `tsconfig.json` с содержимым, показанным в листинге 7.3.

Листинг 7.3. Содержимое файла `tsconfig.json` из папки `types`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Эти параметры конфигурации указывают компилятору TypeScript, что он должен генерировать код для самых последних реализаций JavaScript, используя папку `src` для поиска файлов TypeScript и папку `dist` для генерации вывода результатов. Чтобы настроить NPM для запуска компилятора, добавьте в файл `package.json` строку конфигурации, как показано в листинге 7.4.

Листинг 7.4. Настройка NPM в файле `package.json` из папки `types`

```
{
  "name": "types",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onSuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
```

```
        "tsc-watch": "^6.0.0",
        "typescript": "^5.0.2"
    }
}
```

Для формирования точки входа в проект создайте каталог `types/src` и поместите в него файл `index.ts` с кодом, показанным в листинге 7.5.

Листинг 7.5. Содержимое файла index.ts из папки src

```
console.log("Hello, TypeScript");
```

С помощью консоли выполните в папке `types` команду из листинга 7.6 для запуска компилятора TypeScript.

Листинг 7.6. Запуск компилятора TypeScript

```
npm start
```

Компилятор скомпилирует код в файле `index.ts`, выполнит вывод, а затем перейдет в режим наблюдения, выдав следующий результат:

```
6:43:06 AM - Starting compilation in watch mode...
```

```
6:43:08 AM - Found 0 errors. Watching for file changes.
```

```
Hello, TypeScript
```

7.2. СТАТИЧЕСКИЕ ТИПЫ

Как уже говорилось в главе 4, JavaScript является динамически типизированным. Главное препятствие, которое JavaScript представляет для программистов, привыкших к другим языкам программирования, заключается в том, что **значения** имеют типы, а не переменные. Чтобы вспомнить, как это работает, замените код в файле `index.ts` утверждениями, показанными в листинге 7.7.

Листинг 7.7. Замена содержимого файла index.ts из папки src

```
let myVar;

myVar = 12;
myVar = "Hello";
myVar = true;
```

Тип переменной с именем `myVar` изменяется в зависимости от присвоенного ей значения. Для определения типа можно использовать ключевое слово JavaScript `typeof`, как показано в листинге 7.8.

Листинг 7.8. Отображение типа переменной в файле index.ts из папки src

```
let myVar;
console.log(`${myVar} = ${typeof myVar}`);
myVar = 12;
console.log(`${myVar} = ${typeof myVar}`);
myVar = "Hello";
console.log(`${myVar} = ${typeof myVar}`);
myVar = true;
console.log(`${myVar} = ${typeof myVar}`);
```

После сохранения изменений и при выполнении скомпилированного кода вы увидите следующий результат:

```
undefined = undefined
12 = number
Hello = string
true = boolean
```

Первый оператор в листинге 7.8 определяет переменную без присвоения ей значения, а это означает, что ее тип `undefined`. Переменная, тип которой `undefined`, всегда будет иметь значение `undefined`, как видно из вывода.

Значение `12` — это тип `number`, и при присвоении этого значения тип данных переменной меняется. Значение `Hello` — это `string`, а `false` — `boolean`; тип данных можно увидеть по мере присвоения переменной каждого значения. JavaScript автоматически определяет тип данных по значению, вам не нужно явно указывать его. В качестве краткой справки в табл. 7.3 описаны встроенные типы, которые предоставляет JavaScript.

Таблица 7.3. Встроенные типы JavaScript

Название	Описание
<code>number</code>	Используется для представления числовых значений
<code>string</code>	Используется для представления текстовых данных
<code>boolean</code>	Логический тип данных, который может иметь два возможных значения — <code>true</code> и <code>false</code>
<code>symbol</code>	Используется для представления уникальных постоянных значений, таких как ключи в коллекциях
<code>null</code>	Этому типу может быть присвоено только значение <code>null</code> . Используется для обозначения несуществующей или недопустимой ссылки
<code>undefined</code>	Используется, когда переменная уже определена, но не было присвоено значение
<code>object</code>	Используется для представления составных значений, образованных из отдельных свойств и значений

Динамические типы обеспечивают гибкость, но могут привести и к проблемам, как показано в листинге 7.9, где код в файле `index.ts` заменен функцией и набором операторов, вызывающих ее.

Листинг 7.9. Определение функции в файле `index.ts` из папки `src`

```
function calculateTax(amount) {
    return amount * 1.2;
}

console.log('${12} = ${calculateTax(12)}');
console.log('${"Hello"} = ${calculateTax("Hello")}');
console.log('${true} = ${calculateTax(true)}');
```

Типы параметров функций также являются динамическими, то есть функция `calculateTax` может принимать значения любого типа. Операторы, следующие за функцией, вызывают ее со значениями типа `number`, `string` и `boolean`, что приводит к следующим результатам при выполнении кода:

```
12 = 14.399999999999999
Hello = NaN
true = 1.2
```

С точки зрения JavaScript в этом примере все правильно. Параметры функции могут принимать значения любого типа, и JavaScript обработал каждый тип именно так, как и должен. Но функция `calculateTax` была написана с предположением, что она будет принимать только числовые значения, поэтому только первый результат имеет смысл. (Второй результат, `NaN`, означает, что это «не число», а третий получен путем приведения `true` к числовому значению 1 и использования его в вычислениях — подробнее о приведении типов в JavaScript рассказано в главе 3.)

Нетрудно понять предположение функции о типе ее параметров, когда вы видите код вместе с использующими ее операторами, но гораздо сложнее, когда функция написана другим программистом и находится глубоко внутри сложного проекта или пакета.

7.2.1. Создание статического типа с помощью аннотации типа

Большинство разработчиков привыкли к статическим типам. Функция статических типов в TypeScript делает предположения о типах явными и позволяет компилятору сообщать об ошибке при работе с разными типами данных. Статические типы определяются с помощью *аннотаций типов*, как показано в листинге 7.10.

Листинг 7.10. Использование аннотации типа в файле `index.ts` из папки `src`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

console.log(`${12} = ${calculateTax(12)}`);
console.log(`"${Hello}" = ${calculateTax("Hello")}`);
console.log(`#${true} = ${calculateTax(true)}`);
```

В листинге 7.10 присутствуют две аннотации, определенные с помощью двоеточия, за которым следует статический тип, как показано на рис. 7.1.

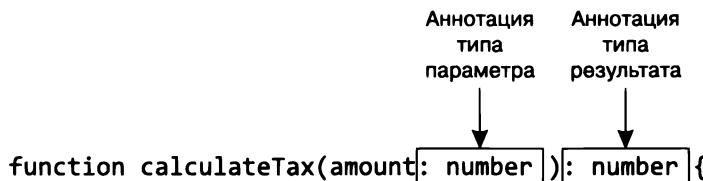


Рис. 7.1. Применение аннотаций типов

Аннотация типа параметра функции указывает компилятору, что функция принимает только числовые значения. Аннотация, следующая за сигнатурой функции, указывает на тип результата и сообщает компилятору, что функция возвращает только числовые значения.

При компиляции кода компилятор TypeScript анализирует типы данных значений, передаваемых в функцию `calculateTax`, обнаруживает, что некоторые значения имеют неправильный тип, и выдает следующие сообщения об ошибках:

```
src/index.ts(6,42): error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
src/index.ts(7,39): error TS2345: Argument of type 'boolean' is not assignable to parameter of type 'number'.
```

СОВЕТ

Вы также можете увидеть предупреждения в своем редакторе кода, если он хорошо поддерживает TypeScript. Я пользуюсь Visual Studio Code для разработки на TypeScript, и он выделяет проблемы прямо в окне редактора.

Аннотации типов могут также применяться к переменным и константам, как показано в листинге 7.11.

Листинг 7.11. Применение аннотаций к переменным в файле index.ts из папки src

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

let price: number = 100;
let taxAmount: number = calculateTax(price);
let halfShare: number = taxAmount / 2;

console.log('Full amount in tax: ${taxAmount}');
console.log('Half share: ${halfShare}');
```

Аннотации указываются после имени переменной с использованием двоеточия и типа, как и в случае с аннотациями, применяемыми к функции. Все три переменные в листинге 7.11 аннотированы, чтобы сообщить компилятору, что они будут использоваться для значений `number`, в результате чего при выполнении кода получится:

```
Full amount in tax: 120
Half share: 60
```

7.2.2. Использование неявно определенных статических типов

Компилятор TypeScript способен выводить типы, что означает возможность использования статических типов без применения аннотаций, как показано в листинге 7.12.

Листинг 7.12. Опора на неявные типы в файле index.ts из папки src

```
function calculateTax(amount: number) {
    return amount * 1.2;
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log('Full amount in tax: ${taxAmount}');
console.log('Half share: ${halfShare}');
```

Компилятор TypeScript может определить тип переменной `price` на основе присвоенного ей литерального значения, которое присваивается при ее определении. Так как компилятор знает, что `100` — это числовое значение, он обрабатывает переменную `price` так, как если бы она была определена с помощью аннотации `number`. Это означает, что это приемлемое значение для использования ее в качестве аргумента функции `calculateTax`.

Компилятор также способен вывести результат работы функции `calculateTax`, поскольку знает, что функция принимает только числовые параметры; `1.2` — числовое значение, и результат умножения двух числовых значений будет `number`.

Результат функции присваивается переменной `taxAmount`, которую компилятор также определяет как `number`. Наконец, компилятор знает тип, получаемый оператором деления двух числовых значений, и может вывести тип переменной `halfShare`.

Компилятор TypeScript не выводит никаких предупреждений, когда типы используются правильно, и легко забыть о том, что код проверяется. Чтобы увидеть, что происходит при несовпадении типов, измените функцию в файле `index.ts`, как показано в листинге 7.13.

Листинг 7.13. Изменение в файле index.ts из папки src

```
function calculateTax(amount: number) {
    return (amount * 1.2).toFixed(2);
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log('Full amount in tax: ${taxAmount}');
console.log('Half share: ${halfShare}');
```

Метод `toFixed` форматирует числовые значения таким образом, чтобы они имели фиксированное количество цифр после десятичной точки. Результатом метода `toFixed` является строка, которая изменяет тип возвращаемого значения функцией `calculateTax` на `string`. Когда компилятор TypeScript обрабатывает цепочку типов, он видит, что оператор деления применяется к строке и числу:

```
...
let halfShare = taxAmount / 2;
...
```

Это легальный JavaScript, и он будет обрабатываться приведением типов, как описано в главе 3. В данном случае строковое значение преобразуется в число, а результатом станет либо деление двух числовых значений, либо `Nan`, если строковое значение не может быть преобразовано.

В TypeScript автоматическое приведение типов ограничено, и компилятор сообщает об ошибке вместо того, чтобы пытаться преобразовать значения:

```
src/index.ts(7,17): error TS2362: The left-hand side of an arithmetic
operation must be of type 'any', 'number', 'bigint' or an enum type.
```

Компилятор TypeScript не предотвращает использование функциональности типов JavaScript, но генерирует ошибки, когда обнаруживает потенциальные проблемы.

В некоторых случаях, особенно когда вы только начинаете работать с TypeScript, могут возникать ошибки, связанные с тем, что компилятор выводит типы не так, как вы ожидаете. Почти в каждом случае компилятор будет прав, но есть полезная функция компилятора, которую можно включить для выявления типов, используемых в коде, как показано в листинге 7.14.

Листинг 7.14. Настройка компилятора в файле tsconfig.json из папки types

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true
  }
}
```

Параметр `declaration` указывает компилятору генерировать файлы, содержащие информацию о типах, вместе с создаваемым JavaScript-кодом. Эти файлы подробно описаны в главе 15, а пока достаточно знать, что они помогают идентифицировать типы, выведенные компилятором, хотя это и не является их целевым назначением. Изменения в конфигурации вступят в силу при следующей компиляции. Чтобы запустить компиляцию, добавьте в файл `index.ts` утверждение, показанное в листинге 7.15, и сохраните изменения.

Листинг 7.15. Добавление утверждения в файл index.ts из папки src

```
function calculateTax(amount: number) {
  return (amount * 1.2).toFixed(2);
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log('Price: ${price}');
console.log('Full amount in tax: ${taxAmount}');
console.log('Half share: ${halfShare}');
```

При запуске компилятора в папке `dist` будет создан файл с именем `index.d.ts`, в котором содержится следующее:

```
declare function calculateTax(amount: number): string;
declare let price: number;
declare let taxAmount: string;
declare let halfShare: number;
```

Назначение ключевого слова `declare` и самого файла описано в главе 15, но данный файл показывает типы, которые компилятор определил для операторов в листинге 7.15. Из этого следует, что возвращаемые типы для функции `calculateTax` и переменной `taxAmount` являются строковыми. При возникновении ошибки компилятора просмотр файлов, созданных при значении параметра `declaration` — `true`, может оказаться полезным, особенно если явная причина неочевидна.

7.2.3. Тип `any`

TypeScript не мешает вам использовать гибкость системы типов JavaScript, но пытается предотвратить их случайное использование. Чтобы разрешить использовать все типы в качестве параметров и результатов функций или иметь возможность присваивать все типы переменным и константам, в TypeScript предусмотрен тип `any`, как показано в листинге 7.16.

Листинг 7.16. Использование типа `any` в файле `index.ts` из папки `src`

```
function calculateTax(amount: any): any {
    return (amount * 1.2).toFixed(2);
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log('Price: ${price}');
console.log('Full amount in tax: ${taxAmount}');
console.log('Half share: ${halfShare}');
```

Эти аннотации указывают компилятору, что параметр `amount` может принимать любое значение, а результат функции может быть любого типа. Использование типа `any` не позволяет компилятору выдать ошибку, как в случае из листинга 7.15, поскольку теперь он больше не проверяет, может ли результат функции `calculateTax` быть использован с оператором деления. Код успешно выполнится, поскольку JavaScript автоматически преобразует операнды деления в значения типа `number`. Таким образом, строка, возвращаемая функцией `calculateTax`, преобразуется в число, что приведет к следующему выводу:

```
Price: 100
Full amount in tax: 120.00
Half share: 60
```

При использовании типа `any` важно осознавать ответственность за корректное применение типов в вашем коде так же, как и при работе с чистым JavaScript. В листинге 7.17 функция `calculateTax` изменена таким образом, чтобы она добавляла символ валюты к своему результату.

Листинг 7.17. Изменение результата функции в файле index.ts из папки src

```
function calculateTax(amount: any): any {
    return '$${(amount * 1.2).toFixed(2)}';
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log('Price: ${price}');
console.log('Full amount in tax: ${taxAmount}');
console.log('Half share: ${halfShare}');
```

Результат функции не может быть преобразован в числовое значение, поэтому при ее выполнении код выдает следующее:

```
Price: 100
Full amount in tax: $120.00
Half share: NaN
```

Одним из последствий использования `any` является то, что его можно присвоить всем остальным типам, не вызывая предупреждения компилятора (листинг 7.18).

Листинг 7.18. Присвоение типа `any` в файле index.ts из папки src

```
function calculateTax(amount: any): any {
    return '$${(amount * 1.2).toFixed(2)}';
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

console.log('Price: ${price}');
console.log('Full amount in tax: ${taxAmount}');
console.log('Half share: ${halfShare}');

let newResult: any = calculateTax(200);
let myNumber: number = newResult;
console.log('Number value: ${myNumber.toFixed(2)}');
```

Значение `any`, присвоенное переменной `newResult`, присваивается числу, не вызывая предупреждения компилятора. Метод `calculateTax` во время выполнения возвращает строку, в результате чего возникает ошибка при вызове метода `toFixed` в последнем операторе листинга 7.18:

```
console.log('Number value: ${myNumber.toFixed(2)}');
^
TypeError: myNumber.toFixed is not a function
```

Компилятор полагает, что значение `any` может рассматриваться как число, что приводит к несоответствию типов во время выполнения программы. Тип `any` позволяет в полной мере использовать возможности типов JavaScript, что может быть полезно, однако это может повлечь за собой неожиданные результаты, когда типы принудительно приводятся к конкретному типу во время выполнения программы.

СОВЕТ

TypeScript также предоставляет тип `unknown`, чтобы обеспечить явный доступ к возможностям динамических типов, ограничивая при этом случайное использование, как описано в разделе 7.6.

Неявное определение типов `any`

Компилятор TypeScript будет использовать `any`, если он присваивает типы неявно и не может определить более конкретный тип для использования. Это облегчает выборочное применение TypeScript в существующем JavaScript-проекте и упрощает работу со сторонними пакетами JavaScript. В листинге 7.19 удалена аннотация типа из параметра `calculateTax`.

Листинг 7.19. Удаление аннотации в файле index.ts из папки src

```
function calculateTax(amount): any {
    return '$${(amount * 1.2).toFixed(2)}';
}

let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;

let personVal = calculateTax("Bob");

console.log('Price: ${price}');
console.log('Full amount in tax: ${taxAmount}');
console.log('Half share: ${halfShare}');
console.log('Name: ${personVal}');
```

Компилятор использует неявный тип `any` для параметра функции, поскольку не может определить более подходящий тип для использования. Таким образом, при вызове функции со строковым аргументом компилятор не выдаст ошибку, а выведет следующий результат:

```
Price: 100
Full amount in tax: $120.00
Half share: NaN
Name: $NaN
```

Подтвердить неявное использование `any` можно, просмотрев содержимое файла `index.d.ts` в папке `dist`, который будет включать следующее описание функции `calculateTax`:

```
...
declare function calculateTax(amount: any): any;
...
```

Отключение неявных типов `any`

Явное использование `any` позволяет избежать проверки типов, что иногда полезно при осторожном применении. Однако разрешение компилятору использовать `any` неявно создает проблемы при проверке типов, которые вы можете даже не заметить и которые могут свести на нет все преимущества от использования TypeScript.

Хорошей практикой считается запрет неявного использования `any` путем установки параметра компилятора `noImplicitAny`, как показано в листинге 7.20. (Неявное использование `any` также отключается при включении параметра `strict`.)

Листинг 7.20. Настройка компилятора в файле tsconfig.json из папки types

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "noImplicitAny": true
  }
}
```

Сохраните изменения в конфигурационном файле компилятора, и код будет перекомпилирован со следующей ошибкой:

```
src/index.ts(1,23): error TS7006: Parameter 'amount' implicitly has an 'any' type.
```

Компилятор выдаст подобное предупреждение, когда не может вывести более конкретный тип, хотя это не препятствует явному использованию типа `any`.

7.3. ОБЪЕДИНЕНИЯ ТИПОВ

На одном конце спектра безопасности типов находится функция `any`, предоставляющая полную свободу. На другом — аннотации типов для конкретного типа, которые ограничивают диапазон допустимых значений. Между этими двумя крайностями в TypeScript располагаются *объединения типов*, которые позволяют объединять несколько типов данных в один. В листинге 7.21 определена функция, возвращающая различные типы данных, с использованием аннотации типа с объединением для описания результата компилятору.

Листинг 7.21. Использование объединения типов в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean): string | number {
  const calcAmount = amount * 1.2;
  return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxNumber = calculateTax(100, false);
let taxString = calculateTax(100, true);
```

Тип, возвращаемый функцией `calculateTax`, представляет собой объединение типов `string` и `number`, которое обозначается символом `|` между наименованиями типов (рис. 7.2). Объединение в листинге 7.21 использует два типа, но вы можете объединить столько типов, сколько вам необходимо.

Важно понимать, что объединение типов рассматривается как самостоятельный тип, характеристиками которого являются пересечения отдельных типов. Это означает, что, например, тип переменной `taxNumber` в листинге 7.21 — `string | number`, а не `number`, несмотря на то что функция `calculateTax` возвращает

число, если аргумент `boolean` равен `false`. Чтобы подчеркнуть влияние типа объединения, в листинге 7.22 типы переменных указаны в явном виде.

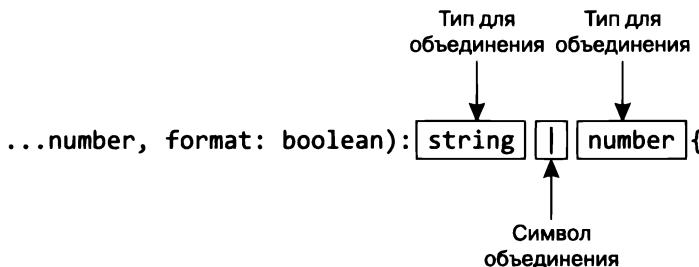


Рис. 7.2. Определение объединения типов

Листинг 7.22. Объявление типов объединений в явном виде в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxNumber: string | number = calculateTax(100, false);
let taxString: string | number = calculateTax(100, true);

console.log('Number Value: ${taxNumber.toFixed(2)}');
console.log('String Value: ${taxString.charAt(0)}');
```

Вы можете использовать только свойства и методы, определенные всеми типами в объединении, что полезно для сложных типов (как будет описано в главе 10), но ограничено небольшим общим API, представленным примитивными значениями. Единственным методом, общим для типов `number` и `string`, которые используются в объединении в листинге 7.22, является метод `toString`, как показано на рис. 7.3.

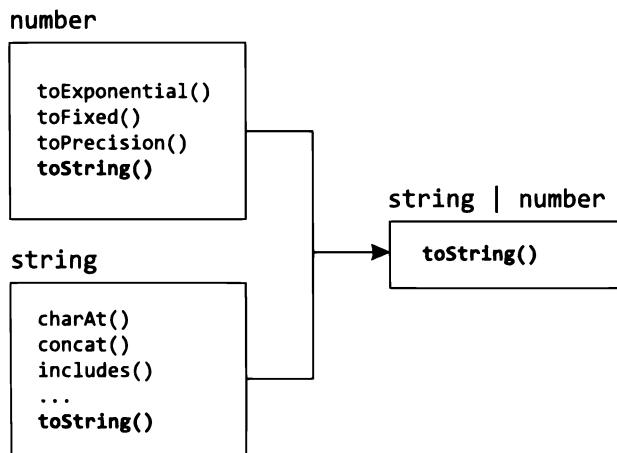


Рис. 7.3. Эффект объединения типов

Это означает, что другие методы, определенные типами `number` и `string`, использовать нельзя, а применение методов `toFixed` и `charAt` в листинге 7.22 приводит к следующим сообщениям компилятора:

```
src/index.ts(9,40): error TS2339: Property 'toFixed' does not exist on type
'string | number'.
  Property 'toFixed' does not exist on type 'string'.
src/index.ts(10,40): error TS2339: Property 'charAt' does not exist on type
'string | number'.
  Property 'charAt' does not exist on type 'number'.
```

7.4. УТВЕРЖДЕНИЯ ТИПА

Утверждение типа указывает компилятору TypeScript, что значение должно рассматриваться как определенный тип, что называется *сужением типа*. Утверждение типа — это один из способов сужения типа из объединения, как показано в листинге 7.23.

Листинг 7.23. Практическое применение утверждений типов в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean): string | number {
  const calcAmount = amount * 1.2;
  return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;

console.log('Number Value: ${taxNumber.toFixed(2)})');
console.log('String Value: ${taxString.charAt(0)})');
```

Тип утверждается с помощью ключевого слова `as`, за которым следует требуемый тип (рис. 7.4).

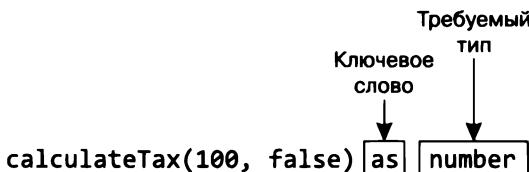


Рис. 7.4. Утверждение типа

В листинге ключевое слово `as` используется для того, чтобы сообщить компилятору, что значение, присвоенное переменной `taxNumber`, является числом, а значение, присвоенное переменной `taxString`, — строкой:

```
...
let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;
...
```

ВНИМАНИЕ

Утверждение типа не выполняет никаких преобразований, оно лишь сообщает компилятору, какой тип он должен применить к значению для проверки типа.

Когда тип утверждается таким образом, TypeScript использует утвержденный тип в качестве типа переменной. А это означает, что выделенные утверждения в листинге 7.23 эквивалентны выражениям ниже:

```
...
let taxNumber: number = calculateTax(100, false) as number;
let taxString: string = calculateTax(100, true) as string;
...
```

Утверждения типа выбирают конкретный тип из объединения, а значит, могут быть использованы методы и свойства, доступные для этого типа, что позволяет избежать ошибок, приведенных в листинге 7.22, и получить следующий результат:

```
Number Value: 120.00
String Value: $
```

7.4.1. Утверждение к неожиданному типу

Компилятор проверяет, что тип, указанный в утверждении, соответствует ожидаемому. Например, при использовании утверждения из объединения типов оно должно относиться к одному из типов, входящих в это объединение. Чтобы увидеть, что происходит при утверждении типа, который компилятор не ожидает, добавьте в файл index.ts код из листинга 7.24.

Листинг 7.24. Утверждение неожиданного типа в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;
let taxBoolean = calculateTax(100, false) as boolean;

console.log('Number Value: ${taxNumber.toFixed(2)}');
console.log('String Value: ${taxString.charAt(0)}');
console.log('Boolean Value: ${taxBoolean}');
```

Утверждение типа указывает компилятору, что значение `string | number` должно обрабатываться как `boolean`. Компилятор знает, что `boolean` не является одним из типов в объединении, и при компиляции выдает следующую ошибку:

```
src/index.ts(8,18): error TS2352: Conversion of type 'string | number' to
type 'boolean' may be a mistake because neither type sufficiently overlaps
with the other. If this was intentional, convert the expression to
'unknown' first.
```

Type 'number' is not comparable to type 'boolean'.

В большинстве ситуаций следует просмотреть типы данных и утверждение типа и устранить проблему, расширив объединение типов или установив утверждение другого типа. Однако можно принудительно выполнить утверждение и отключить предупреждение компилятора, выполнив сначала утверждение к `any`, а затем к требуемому типу, как показано в листинге 7.25. (Ошибка компилятора связана с типом `unknown`, подробнее о котором — в разделе 7.6.)

Листинг 7.25. Утверждение неожиданного типа в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;
let taxBoolean = calculateTax(100, false) as any as boolean;

console.log('Number Value: ${taxNumber.toFixed(2)}');
console.log('String Value: ${taxString.charAt(0)}');
console.log('Boolean Value: ${taxBoolean}');
```

Этот дополнительный шаг не позволяет компилятору выдавать предупреждение об изменении типа и обрабатывает результат работы функции как логическое значение. Однако, как отмечалось ранее, утверждения влияют только на процесс проверки типов и не выполняют преобразование типов, что видно по результатам, получаемым при компиляции кода:

```
Number Value: 120.00
String Value: $
Boolean Value: 120
```

Результат, выдаваемый функцией, был описан компилятору как объединение `string | number` и утверждается как `boolean`. Но при выполнении кода функция возвращает `number`, значение которого записывается в консоль.

7.5. ЗАЩИТА ТИПА

Для примитивных значений можно использовать ключевое слово `typeof` для проверки наличия конкретного типа без необходимости утверждения типа, как показано в листинге 7.26.

Листинг 7.26. Использование защиты типа в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

if (typeof taxValue === "number") {
    console.log('Number Value: ${taxValue.toFixed(2)}');
```

```
 } else if (typeof taxValue === "string") {
     console.log('String Value: ${taxValue.charAt(0)}');
}
```

Для проверки типа к значению применяется ключевое слово `typeof`, в результате чего получается строка, которую можно сравнить с именами примитивных типов JavaScript, таких как `number` и `boolean`.

ПРИМЕЧАНИЕ

Ключевое слово `typeof` может быть использовано только с примитивными типами JavaScript. Для различия объектов требуется другой подход, описанный в главах 3 и 10.

Компилятор не реализует ключевое слово `typeof`, так как оно является частью спецификации JavaScript. Вместо этого он полагается на то, что утверждения в условном блоке будут выполнены в среде выполнения только в том случае, если проверяемое значение имеет указанный тип. Это знание позволяет компилятору рассматривать значение как проверяемый тип. Например, первый тест в листинге 7.26 предназначен для `number`:

```
...
if (typeof taxValue === "number") {
    console.log('Number Value: ${taxValue.toFixed(2)})');
}
...
```

Компилятор TypeScript знает, что утверждения внутри блока `if` будут выполнены, только если `taxValue` является числом, и позволяет использовать метод `toFixed` типа `number` без необходимости утверждения типа, что приводит к следующему результату при компиляции кода:

```
Number Value: 120.00
```

Компилятор умеет распознавать операторы защиты типов, даже если они не находятся в обычном блоке `if...else`. Код из листинга 7.27 дает тот же результат, что и из листинга 7.26, но для различия типов используется оператор `switch`. Внутри каждого блока компилятор обрабатывает `taxValue` так, как будто он был определен только с типом, выбранным оператором `case`.

Листинг 7.27. Защита типов в операторе switch в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log('Number Value: ${taxValue.toFixed(2)})');
        break;
    case "string":
        console.log('String Value: ${taxValue.charAt(0)})');
        break;
}
```

7.5.1. Тип never

TypeScript предоставляет тип `never` для ситуаций, когда защита типа обрабатывает все возможные типы значений. Например, в листинге 7.27 оператор `switch` является защищой типа для типов `number` и `string`, которые являются единственными типами, возвращаемыми из функции в объединении `string | number`. После обработки всех возможных типов компилятор разрешит присвоить значение только типу `never`, как показано в листинге 7.28.

Листинг 7.28. Использование типа `never` в файле `index.ts` из папки `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log('Number Value: ${taxValue.toFixed(2)}');
        break;
    case "string":
        console.log('String Value: ${taxValue.charAt(0)}');
        break;
    default:
        let value: never = taxValue;
        console.log('Unexpected type for value: ${value}');
}
}
```

Если выполнение достигает предложения `default` оператора `switch`, значит, что-то пошло не так. TypeScript предоставляет тип `never`, чтобы исключить случайное использование значения после того, как были использованы механизмы защиты типа для исчерпывающего сужения значения до всех его возможных типов.

7.6. ТИП UNKNOWN

В подразделе 7.2.3 я объяснил, что значение `any` может быть присвоено всем остальным типам, что создает пробел в проверке типов компилятором. TypeScript также поддерживает тип `unknown`, который является более безопасной альтернативой `any`. Значение `unknown` может быть присвоено только типу `any` или самому себе, если не используется утверждение типа или защита типа. В листинге 7.29 повторяются утверждения из примера, показывающего поведение типа `any`, но вместо него используется `unknown`.

Листинг 7.29. Использование типов `any` и `unknown` в файле `index.ts` из папки `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}
```

```

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log('Number Value: ${taxValue.toFixed(2)}');
        break;
    case "string":
        console.log('String Value: ${taxValue.charAt(0)}');
        break;
    default:
        let value: never = taxValue;
        console.log('Unexpected type for value: ${value}');
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult;
console.log('Number value: ${myNumber.toFixed(2)}');

```

Значение `unknown` не может быть присвоено другому типу без утверждения типа, поэтому компилятор при компиляции кода выдает следующую ошибку:

```
src/index.ts(18,5): error TS2322: Type 'unknown' is not assignable to type 'number'.
```

В листинге 7.30 используется утверждение типа, чтобы отменить предупреждение и указать компилятору на присвоение значения `unknown` как `number`.

Листинг 7.30. Утверждение значения `unknown` в файле `index.ts`

```

function calculateTax(amount: number, format: boolean): string | number {
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log('Number Value: ${taxValue.toFixed(2)}');
        break;
    case "string":
        console.log('String Value: ${taxValue.charAt(0)}');
        break;
    default:
        let value: never = taxValue;
        console.log('Unexpected type for value: ${value}');
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult as number;
console.log('Number value: ${myNumber.toFixed(2)}');

```

В отличие от предыдущего примера значение `unknown` действительно является числом, поэтому код не генерирует ошибку среди выполнения и выдает следующий результат:

```
Number Value: 120.00
Number value: 240.00
```

7.7. ТИП NULL

В системе статических типов TypeScript есть дыра: типы JavaScript `null` и `undefined`. Типу `null` может быть присвоено только значение `null`, и он используется для представления чего-то несуществующего или недопустимого. Типу `undefined` можно присвоить только значение `undefined`, и он используется, когда переменная уже определена, но еще не присвоено значение.

Проблема в том, что по умолчанию TypeScript рассматривает `null` и `undefined` как допустимые значения для всех типов. Причина этого заключается в удобстве, поскольку большая часть существующего JavaScript-кода, который может потребоваться для интеграции в приложение, использует эти значения как часть своей нормальной работы, что приводит к несогласованности в проверке типов, как показано в листинге 7.31.

Листинг 7.31. Использование типов `null` в файле `index.ts` из папки `src`

```
function calculateTax(amount: number, format: boolean): string | number {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue: string | number = calculateTax(0, false);

switch (typeof taxValue) {
    case "number":
        console.log('Number Value: ${taxValue.toFixed(2)}');
        break;
    case "string":
        console.log('String Value: ${taxValue.charAt(0)}');
        break;
    default:
        let value: never = taxValue;
        console.log('Unexpected type for value: ${value}');
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult as number;
console.log('Number value: ${myNumber.toFixed(2)})');
```

В изменении `calculateTax` показано типичное применение `null`, где он используется в качестве результата, если значение параметра `amount` равно нулю, что указывает на недопустимое условие. Тип результата для функции и тип переменной `taxValue` — `string | number`. Однако в JavaScript изменение значения, присвоенного переменной, может изменить ее тип, что и происходит в примере: второй вызов функции `calculateTax` возвращает `null`, что изменяет тип `taxValue` на `null`. Когда операторы защиты типа проверяют тип переменной, они не могут сузить ее тип до одного из тех, которые входят в объединение `string | number`, и выдают следующий результат:

```
Unexpected type for value: null
Number value: 240.00
```

В обычных обстоятельствах компилятор сообщит об ошибке, если значение одного типа присваивается переменной другого типа, но не в этом случае, поскольку компилятор позволяет рассматривать `null` и `undefined` как значения для всех типов.

ПРИМЕЧАНИЕ

Помимо несоответствия типов, значения типа `null` во время выполнения могут приводить к ошибкам во время выполнения, которые трудно обнаружить на этапе разработки и с которыми часто сталкиваются пользователи. Например, в листинге 7.31 потребители функции `calculateTax` не могут легко узнать, что может быть возвращено значение `null`, и понять, когда это может произойти. В примере легко увидеть значение `null` и причины его использования, но гораздо сложнее сделать то же самое в реальном проекте или в стороннем пакете.

7.7.1. Ограничение присвоений значения `null`

Использование `null` и `undefined` можно ограничить, включив настройку компилятора `strictNullChecks`, как показано в листинге 7.32 (Этот параметр также включается с помощью опции `strict`.)

Листинг 7.32. Включение строгой проверки на `null` в файле `tsconfig.json` из папки `types`

```
{  
  "compilerOptions": {  
    "target": "ES2022",  
    "outDir": "./dist",  
    "rootDir": "./src",  
    "declaration": true,  
    "noImplicitAny": true,  
    "strictNullChecks": true  
  }  
}
```

При значении `true` этот параметр не разрешает компилятору присваивать другим типам значения `null` или `undefined`. Сохраните изменения в файле конфигурации, и компилятор перекомпилирует файл `index.ts` и выдаст следующее сообщение:

```
src/index.ts(3,9): error TS2322: Type 'null' is not assignable to type  
'string | number'.
```

Изменение конфигурации предписывает компилятору выдавать ошибку при присвоении значений `null` или `undefined` другому типу. В нашем примере ошибка возникает потому, что значение `null`, возвращаемое функцией `calculateTax`, не является одним из типов в объединении, описывающем результат функции.

Для устранения ошибки можно переписать функцию так, чтобы она не задействовала `null`, или расширить объединение типов, используемое для описания ее результата, включив в него значение `null`, как в листинге 7.33.

Листинг 7.33. Расширение объединения типов в файле index.ts из папки src

```

function calculateTax(amount: number, format: boolean)
    : string | number | null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue: string | number | null = calculateTax(0, false);

switch (typeof taxValue) {
    case "number":
        console.log('Number Value: ${taxValue.toFixed(2)}');
        break;
    case "string":
        console.log('String Value: ${taxValue.charAt(0)}');
        break;
    default:
        if (taxValue === null) {
            console.log("Value is null");
        } else {
            console.log(typeof taxValue);
            let value: never = taxValue;
            console.log('Unexpected type for value: ${value}');
        }
}

```

Расширение объединения типов делает очевидным тот факт, что функция может возвращать значения `null`, гарантируя, что код, использующий функцию, понимает, что необходимо работать со значениями `string`, `number` или `null`. Как объяснялось в главе 3, использование `typeof` для значений `null` возвращает `object`, поэтому защита от `null` осуществляется с помощью явной проверки значения, которую компилятор TypeScript воспринимает как защиту типа. Код в листинге 7.33 при выполнении приводит к следующему результату:

`Value is null`

7.7.2. Удаление `null` из объединения с помощью утверждения

Помните, что объединения представляют пересечение API каждого типа. Значения `null` и `undefined` не представляют никаких свойств или методов, то есть значения для объединений типов, допускающих значение `null`, не могут использоваться напрямую. Даже если отличные от `null` типы имеют пересечение полезных свойств или методов (примеры которых будут приведены в последующих главах), использование этих значений требует специального обращения. Утверждение, отличное от `null`, сообщает компилятору, что значение не равно `null`, что удаляет `null` из объединения типов и позволяет использовать пересечение других типов, как показано в листинге 7.34.

ВНИМАНИЕ

Утверждение, отличное от `null`, следует использовать только в том случае, если известно, что значение `null` не может возникнуть. Если применить утверждение и получить значение `null`, то возникнет ошибка во время выполнения. Более безопасным подходом является использование защиты типа, как описано в следующем разделе.

Листинг 7.34. Использование утверждения, отличного от `null`, в файле `index.ts` из папки `src`

```
function calculateTax(amount: number, format: boolean)
    : string | number | null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue: string | number = calculateTax(100, false)!;

switch (typeof taxValue) {
    case "number":
        console.log('Number Value: ${taxValue.toFixed(2)}');
        break;
    case "string":
        console.log('String Value: ${taxValue.charAt(0)}');
        break;
    default:
        if (taxValue === null) {
            console.log("Value is null");
        } else {
            console.log(typeof taxValue);
            let value: never = taxValue;
            console.log('Unexpected type for value: ${value}');
        }
}
```

Чтобы обозначить, что значение является отличным от `null`, используется символ `!`, как показано на рис. 7.5. Такое утверждение говорит компилятору, что результат работы функции `calculateTax` не будет `null`. Это позволяет присвоить его переменной `taxValue`, тип которой `string | number`.

Код из листинга 7.34 при компиляции и выполнении выводит следующее:

Number Value: 120.00

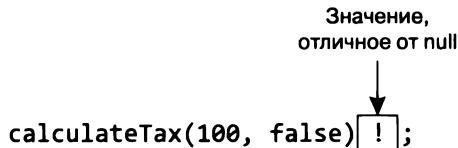


Рис. 7.5. Утверждение значения, отличного от `null`

7.7.3. Удаление null из объединения с помощью защиты типа

Альтернативным подходом является фильтрация значений `null` или `undefined` с помощью защиты типа, как показано в листинге 7.35. Преимущество этого метода заключается в проверке значения во время выполнения программы.

Листинг 7.35. Удаление значений `null` с помощью защиты типа в файле `index.ts` из папки `src`

```
function calculateTax(amount: number, format: boolean)
    : string | number | null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue: string | number | null = calculateTax(100, false);
if (taxValue !== null) {
    let nonNullTaxValue: string | number = taxValue;
    switch (typeof taxValue) {
        case "number":
            console.log('Number Value: ${taxValue.toFixed(2)}');
            break;
        case "string":
            console.log('String Value: ${taxValue.charAt(0)}');
            break;
    }
} else {
    console.log("Value is not a string or a number");
}
```

Компилятор знает, что проверка значения на `null` показывает, что оно может быть обработано как тип объединения `string | number`, не допускающий значения `null`, с помощью блока кода `if`. (Компилятор также знает, что `taxValue` может быть `null` только в блоке кода `else`.) При компиляции и выполнении код из листинга 7.35 выдаст следующий результат:

```
Number Value: 120.00
```

7.7.4. Утверждение определения присваивания

Если опция `strictNullChecks` включена, то компилятор сообщит об ошибке, если переменная используется до того, как ей присвоится значение. Это полезная функция, но бывают случаи, когда значение присваивается способом, невидимым для компилятора, как показано в листинге 7.36.

ВНИМАНИЕ

В листинге 7.36 используется встроенная функция JavaScript `eval` для выполнения строки в качестве оператора кода. Функция `eval` считается небезопасной и не должна использоваться в реальных проектах.

Листинг 7.36. Использование неназначенной переменной в файле index.ts из папки src

```
function calculateTax(amount: number, format: boolean)
    : string | number | null {
  if (amount === 0) {
    return null;
  }
  const calcAmount = amount * 1.2;
  return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue: string | number | null;
eval("taxValue = calculateTax(100, false)");

if (taxValue !== null) {
  let nonNullTaxValue: string | number = taxValue;
  switch (typeof taxValue) {
    case "number":
      console.log('Number Value: ${taxValue.toFixed(2)}');
      break;
    case "string":
      console.log('String Value: ${taxValue.charAt(0)})');
      break;
  }
} else {
  console.log("Value is not a string or a number");
}
```

Функция `eval` принимает строку `string` и выполняет ее как оператор кода. Компилятор TypeScript не может определить действие функции `eval` и не понимает, что она присваивает значение `taxValue`. При компиляции кода компилятор выдает следующие ошибки:

```
src/index.ts(13,5): error TS2454: Variable 'taxValue' is used before being
assigned.
src/index.ts(14,9): error TS2322: Type 'string | number | null' is not
assignable to type 'string | number'.
  Type 'null' is not assignable to type 'string | number'.
src/index.ts(14,44): error TS2454: Variable 'taxValue' is used before being
assigned.
src/index.ts(15,20): error TS2454: Variable 'taxValue' is used before being
assigned.
```

Утверждение определения присваивания сообщает TypeScript, что значение присвоится до того, как переменная будет применена (листинг 7.37).

Листинг 7.37. Использование утверждения определения присваивания в файле index.ts

```
function calculateTax(amount: number, format: boolean)
    : string | number | null {
  if (amount === 0) {
    return null;
  }
  const calcAmount = amount * 1.2;
  return format ? `${calcAmount.toFixed(2)}` : calcAmount;
}
```

```

let taxValue!: string | number | null;
eval("taxValue = calculateTax(100, false)");

if (taxValue !== null) {
    let nonNullTaxValue: string | number = taxValue;
    switch (typeof taxValue) {
        case "number":
            console.log('Number Value: ${taxValue.toFixed(2)}');
            break;
        case "string":
            console.log('String Value: ${taxValue.charAt(0)})');
            break;
    }
} else {
    console.log("Value is not a string or a number");
}

```

Утверждение определения присваивания объявляется символом `!`, который добавляется после имени переменной при ее определении, в отличие от утверждения `non-null`, которое используется в выражениях. Как и в случае с другими утверждениями, вы несете ответственность за присвоение значения. Если использовать утверждение, но не выполнить присваивание, можно столкнуться с ошибкой при выполнении программы. Утверждение, приведенное в листинге 7.37, позволяет скомпилировать код, который выдает следующий результат:

```
Number Value: 120.00
```

РЕЗЮМЕ

В этой главе мы рассмотрели, как можно использовать TypeScript для ограничения системы типов JavaScript путем выполнения проверки типов. Также было показано, как с помощью аннотаций типов указать допустимые типы переменных и как компилятор может вывести типы из утверждений кода. Помимо прочего, я объяснил использование типов `any`, `unknown` и `never`, рассказал об объединении и защите типов, которые ограничивают допустимый диапазон значений.

- Статические типы — это главная особенность TypeScript, делающая систему типов JavaScript более простой в использовании и более предсказуемой для большинства программистов.
- Типы можно явно указывать с помощью аннотации типа или доверить компилятору неявно выводить их из контекста.
- Объединения типов представляют собой комбинации типов, что позволяет переменным принимать значения разных типов, входящих в объединение.
- Утверждения типа сообщают компилятору о точном типе значения, позволяя указать конкретный тип из объединения или переопределить понимание данного типа компилятором.
- Ключевое слово `typeof` в JavaScript может быть использовано в качестве альтернативы утверждению типа для примитивных типов JavaScript.

- Тип `any` используется для обозначения переменной, которой можно присваивать значения любого типа.
- Значения типа `undefined` могут быть присвоены переменным других типов только с помощью утверждения типа или путем присваивания через тип `any`.
- Тип `never` предназначен для предотвращения случайного использования значений, не имеющих ожидаемого типа.
- По умолчанию TypeScript разрешает присваивать значения `null` и `undefined` любой переменной, но это поведение можно изменить, установив конфигурационное свойство компилятора `strictNullChecks` в `true`.

В следующей главе мы более подробно поговорим о функциях в TypeScript.

8

Функции

В этой главе

- ✓ Определение функций со статическими типами данных для параметров и результатов.
- ✓ Работа с необязательными параметрами функций.
- ✓ Определение параметров функций со значениями по умолчанию.
- ✓ Использование rest-параметров для получения нескольких значений аргументов.
- ✓ Перегрузка типов функций.
- ✓ Использование функций утверждения в качестве защиты типов.

Эта глава посвящена тому, как TypeScript применяется к функциям и помогает предотвратить распространенные проблемы при определении функций, как работать с параметрами и выдачей результатов. В табл. 8.1 приведено краткое содержание главы.

В качестве краткой справки в табл. 8.2 перечислены параметры компилятора TypeScript, используемые в данной главе.

Таблица 8.1. Краткое содержание главы

Задача	Решение	Листинг
Разрешить вызов функции, если аргументов меньше, чем параметров	Определите необязательные параметры или параметры со значениями по умолчанию	7, 8
Разрешить вызов функции, если аргументов больше, чем параметров	Используйте rest-параметр	9, 10
Ограничить типы, которые могут быть использованы для значений параметров и результатов	Примените аннотации типов к параметрам или сигнатурам функций	11, 17, 18
Запретить использование значений null в качестве аргументов функций	Включите опцию компилятора strictNullChecks	12–14
Убедиться, что все пути выполнения функции возвращают результат	Включите опцию компилятора noImplicitReturns	15, 16
Описать связь между типами параметров функции и ее результатом	Перегрузите тип функции	19, 20
Описать действие функции утверждения	Используйте ключевое слово assert	21–23

Таблица 8.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода
outDir	Задает каталог, в котором будут размещены файлы JavaScript
rootDir	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
declaration	Создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода. Более подробно эти файлы описаны в главе 15
strictNullChecks	Не допускает присваивания null и undefined в качестве значений для других типов
noImplicitReturns	Требует, чтобы все пути выполнения функции возвращали результат
noUnusedParameters	Выдает предупреждение, если функция определяет неиспользуемые параметры

8.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В этой главе мы продолжим использовать проект `types`, созданный в главе 7. Прежде чем приступить к работе, замените содержимое файла `index.ts` в папке `src` кодом из листинга 8.1.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 8.1. Содержимое файла index.ts из папки src

```
function calculateTax(amount) {
    return amount * 1.2;
}

let taxValue = calculateTax(100);
console.log('Total Amount: ${taxValue}');
```

Закомментируйте параметры компилятора, предотвращающие неявное использование типа `any` и присвоение другим типам значений `null` и `undefined`, как показано в листинге 8.2.

Листинг 8.2. Отключение параметров компилятора в файле tsconfig.json из папки types

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    // "noImplicitAny": true,
    // "strictNullChecks": true
  }
}
```

Откройте новое окно консоли, перейдите в папку `types` и выполните команду из листинга 8.3 для запуска компилятора TypeScript, чтобы он автоматически выполнял код после его компиляции.

Листинг 8.3. Запуск компилятора TypeScript

```
npm start
```

Компилятор скомпилирует код в файле `index.ts`, выполнит вывод, а затем перейдет в режим наблюдения, выдав следующий результат:

```
6:52:41 AM - Starting compilation in watch mode...
```

```
6:52:43 AM - Found 0 errors. Watching for file changes.
Total Amount: 120
```

8.2. ОПРЕДЕЛЕНИЕ ФУНКЦИЙ

TypeScript преобразует функции JavaScript, делая их более предсказуемыми и делая предположения о типе данных явными, что позволяет компилятору проверить их. Файл `index.ts` содержит такую простую функцию:

```
...
function calculateTax(amount) {
    return amount * 1.2;
}
...
...
```

В главе 7 мы рассмотрели, как TypeScript может использовать аннотации типов для функций. В последующих разделах мы вновь обратимся к этим возможностям и поговорим о других методах улучшения функций с помощью TypeScript.

8.2.1. Переопределение функций

Одним из наиболее важных изменений, которые вносит TypeScript, является предупреждение о переопределении функции. В JavaScript функция может быть определена несколько раз и при вызове функции используется последняя реализация. Это приводит к распространенной проблеме для разработчиков, перешедших на JavaScript с другого языка программирования (листинг 8.4).

Листинг 8.4. Переопределение функции в файле `index.ts` из папки `src`

```
function calculateTax(amount) {
    return amount * 1.2;
}

function calculateTax(amount, discount) {
    return calculateTax(amount) - discount;
}

let taxValue = calculateTax(100);
console.log('Total Amount: ${taxValue}');
```

Многие языки поддерживают перегрузку функций, что позволяет определять несколько функций с одним и тем же именем, но с разным количеством параметров или разными типами параметров. Если вы привыкли к подобному подходу, то, скорее всего, код из листинга 8.4 покажется вам естественным и вы можете подумать, что вторая функция `calculateTax` строится на основе первой `calculateTax` для применения скидки.

JavaScript не поддерживает перегрузку функций, и при определении двух функций с одинаковым именем вторая заменяет первую независимо от их параметров. Количество аргументов, передаваемых в функцию, в JavaScript не имеет значения — если параметров больше, чем аргументов, то лишние параметры определяются как `undefined`. Если же аргументов больше, чем параметров, то функция может либо

игнорировать их, либо использовать специальное значение `arguments`, которое предоставляет доступ ко всем аргументам, использованным для вызова функции. При выполнении кода из листинга 8.4 первая функция `calculateTax` будет проигнорирована, а вторая — вызвана, но без значения для второго параметра. Подобная ситуация приведет к рекурсивному вызову функции до полного исчерпания стека вызовов и, как результат, к ошибке.

Чтобы избежать этой проблемы, компилятор TypeScript сообщит об ошибке, если с одним и тем же именем определено более одной функции. Вот сообщения об ошибках, выдаваемые компилятором для кода в листинге 8.4:

```
src/index.ts(1,10): error TS2393: Duplicate function implementation.
src/index.ts(5,10): error TS2393: Duplicate function implementation.
```

Практический эффект от невозможности перегрузки функций заключается в том, что приходится либо использовать разные имена (например, `calculateTax` и `calculateTaxWithDiscount`), либо адаптировать поведение одной функции в зависимости от переданных параметров. Первый подход хорошо работает для сложных групп функций, а для более простых задач предпочтительнее второй. В листинге 8.5 использован второй подход и вся функциональность объединена в одну функцию.

Листинг 8.5. Консолидация функций в файле index.ts из папки src

```
function calculateTax(amount, discount) {
    return (amount * 1.2) - discount;
}

let taxValue = calculateTax(100, 0);
console.log('Total Amount: ${taxValue}');
```

Код в листинге 8.5 при компиляции и выполнении выдает следующий результат:

```
Total Amount: 120
```

8.2.2. Параметры функций

В листинге 8.5 внесены два изменения, необходимые для успешной компиляции кода: удаление дублирующей функции `calculateTax` и добавление ее функциональности к первой функции, а также добавление еще одного аргумента в оператор, вызывающий функцию:

```
...
let taxValue = calculateTax(100, 0);
...
```

TypeScript придерживается более строгого подхода, чем JavaScript, и ожидает, что функции будут вызываться с тем же количеством аргументов, что и параметры. Добавьте в файл `index.ts` утверждения, показанные в листинге 8.6, чтобы увидеть, как компилятор реагирует на разное количество аргументов.

Листинг 8.6. Вызов функции в файле index.ts

```
function calculateTax(amount, discount) {
    return (amount * 1.2) - discount;
}

let taxValue = calculateTax(100, 0);
console.log('2 args: ${taxValue}');
taxValue = calculateTax(100);
console.log('1 arg: ${taxValue}');
taxValue = calculateTax(100, 10, 20);
console.log('3 args: ${taxValue}');
```

Первый новый вызов функции не содержит достаточного количества аргументов, а второй — слишком много. При компиляции кода компилятор выдает следующие ошибки:

```
src/index.ts(7,12): error TS2554: Expected 2 arguments, but got 1.
src/index.ts(9,34): error TS2554: Expected 2 arguments, but got 3.
```

Компилятор настаивает на соответствии аргументов параметрам, чтобы сделать ожидания в коде явными, как и для функций, описанных в главе 7. Рассматривая набор параметров, нелегко определить, как поведет себя функция, если некоторые из них не получат значений. А когда функция вызывается с другим количеством аргументов, сложно определить, намеренно ли это или ошибка. TypeScript решает обе проблемы, требуя наличия аргументов, соответствующих всем параметрам, если только функция не указывает, что она может быть более гибкой, используя возможности, описанные в следующих разделах.

СОВЕТ

При включении опции `noUnusedParameters` компилятор предупредит вас, если функция попытается определить параметры, которые она не использует.

Необязательные параметры

По умолчанию параметры функции обязательны. Однако это можно изменить с помощью необязательных (`optional`) параметров, как показано в листинге 8.7 (вы могли заметить, что оператор, имеющий слишком много аргументов, закомментирован, о чём мы поговорим в следующих разделах).

Листинг 8.7. Определение необязательного параметра в файле index.ts

```
function calculateTax(amount, discount?) {
    return (amount * 1.2) - (discount || 0);
}

let taxValue = calculateTax(100, 0);
console.log('2 args: ${taxValue}');
taxValue = calculateTax(100);
console.log('1 arg: ${taxValue}');
//taxValue = calculateTax(100, 10, 20);
//console.log('3 args: ${taxValue}');
```

Необязательные параметры определяются путем размещения символа ? после имени параметра, как показано на рис. 8.1.

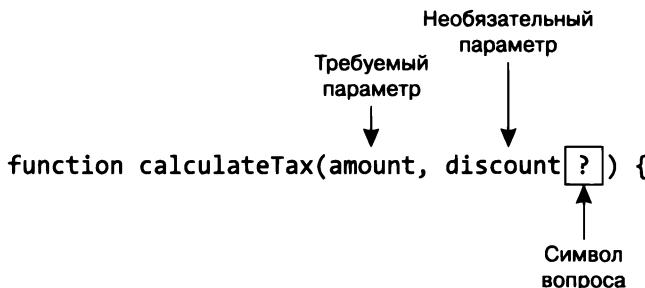


Рис. 8.1. Определение необязательного параметра

ПРИМЕЧАНИЕ

Необязательные параметры должны быть определены после обязательных. Это означает, что, например, в листинге 8.7 мы не можем изменить порядок следования параметров `amount` и `discount`, поскольку `amount` – обязательный параметр, а `discount` – нет.

Вызывающие функцию `calculateTax` могут не указывать значение параметра `discount`, в результате чего функция получит значение `undefined` параметра. Функции, объявляющие необязательные параметры, должны обеспечить возможность работы при отсутствии значений. Функция из листинга 8.7 делает это с помощью логического оператора ИЛИ (||) для преобразования неопределенных значений в ноль, если параметр `discount` не определен:

```

...
return (amount * 1.2) - (discount || 0);
...

```

Параметр `discount` используется так же, как и обязательный параметр, с единственным отличием – функция должна уметь обрабатывать значения `undefined`.

Пользователю функции не нужно предпринимать никаких специальных мер для работы с необязательным параметром. В нашем случае это означает, что функция `calculateTax` может использоваться как с одним, так и с двумя аргументами. Код в листинге 8.7 при выполнении выдает следующий результат:

```

2 args: 120
1 arg: 120

```

Параметр со значением по умолчанию

Если для необязательного параметра предусмотрено значение по умолчанию, которое должно использоваться, то оно может быть применено при определении параметра, как показано в листинге 8.8.

Листинг 8.8. Использование значения параметра по умолчанию в файле index.ts

```

function calculateTax(amount, discount = 0) {
    return (amount * 1.2) - discount;
}

```

```
let taxValue = calculateTax(100, 0);
console.log('2 args: ${taxValue}');
taxValue = calculateTax(100);
console.log('1 arg: ${taxValue}');
//taxValue = calculateTax(100, 10, 20);
//console.log('3 args: ${taxValue}');
```

Параметр, имеющий значение по умолчанию, называется *параметром с инициализацией по умолчанию*. За именем параметра следует оператор присваивания (=) и значение, как показано на рис. 8.2. Обратите внимание, что при определении параметра со значением по умолчанию не используется вопросительный знак.

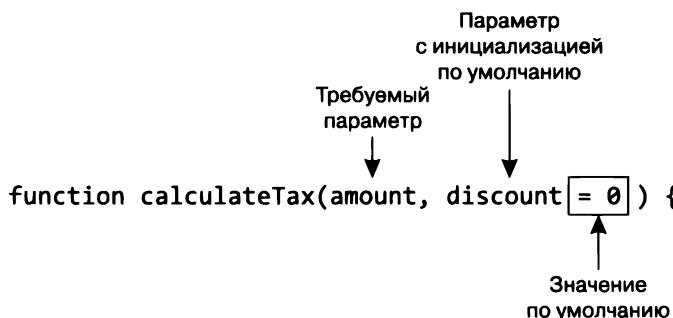


Рис. 8.2. Определение значения параметра по умолчанию

Использование значения по умолчанию означает, что код в функции не должен проверять наличие значений `undefined`, а также то, что значение по умолчанию может быть изменено в одном месте и действовать внутри всей функции.

СОВЕТ

Параметры со значениями по умолчанию все равно являются необязательными, даже если не используется вопросительный знак. Они должны быть определены после обязательных параметров функции.

Код в листинге 8.8 при компиляции и выполнении выдает следующий результат:

```
2 args: 120
1 arg: 120
```

Параметр `rest`

Аналогом необязательных параметров является *параметр rest*, который позволяет функции принимать переменное количество аргументов, объединяя их в массив. Функция может иметь только один параметр `rest`, и он должен быть последним, как показано в листинге 8.9.

Листинг 8.9. Определение параметра `rest` в файле `index.ts`

```
function calculateTax(amount, discount = 0, ...extraFees) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}
```

```
let taxValue = calculateTax(100, 0);
console.log('2 args: ${taxValue}');
taxValue = calculateTax(100);
console.log('1 arg: ${taxValue}');
taxValue = calculateTax(100, 10, 20);
console.log('3 args: ${taxValue}');
```

Для определения параметра rest перед именем параметра ставится многоточие (три точки), как показано на рис. 8.3.



Рис. 8.3. Определение параметра rest

Все аргументы, для которых нет соответствующих параметров, присваиваются rest-параметру, который представляет собой массив. Массив всегда будет инициализирован, но не будет содержать элементов, если не было дополнительных аргументов. Добавление параметра rest означает, что функцию `calculateTax` можно вызывать с одним или несколькими аргументами: первый присваивается параметру `amount`, средний аргумент (если он есть) присваивается параметру `discount`, а все остальные добавляются в массив параметров `extraFees`.

Процесс группировки аргументов в rest-массив параметров происходит автоматически, и никаких специальных действий при вызове функции не требуется. Пользователь функции может задать дополнительные аргументы и отделить их запятыми, как показано в листинге 8.10.

Листинг 8.10. Использование дополнительных аргументов функции в файле index.ts

```
function calculateTax(amount, discount = 0, ...extraFees) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(100, 0);
console.log('2 args: ${taxValue}');
taxValue = calculateTax(100);
console.log('1 arg: ${taxValue}');
taxValue = calculateTax(100, 10, 20);
console.log('3 args: ${taxValue}');
taxValue = calculateTax(100, 10, 20, 1, 30, 7);
console.log('6 args: ${taxValue}');
```

Код в листинге 8.10 при компиляции и выполнении приводит к следующим результатам:

```
2 args: 120
1 arg: 120
3 args: 130
6 args: 168
```

Применение аннотаций типов к параметрам функций

По умолчанию компилятор TypeScript присваивает всем параметрам функции тип `any`, однако с помощью аннотаций можно объявить более конкретные типы. В листинге 8.11 к функции `calculateTax` применены аннотации типов, чтобы гарантировать, что в качестве ее параметров могут выступать только значения типа `number`.

Листинг 8.11. Применение аннотаций типов параметров в файле `index.ts`

```
function calculateTax(amount: number,
    discount: number = 0, ...extraFees: number[]) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(100, 0);
console.log('2 args: ${taxValue}');
taxValue = calculateTax(100);
console.log('1 arg: ${taxValue}');
taxValue = calculateTax(100, 10, 20);
console.log('3 args: ${taxValue}');
taxValue = calculateTax(100, 10, 20, 1, 30, 7);
console.log('6 args: ${taxValue}');
```

Для параметров со значениями по умолчанию аннотация типа располагается перед присвоением значения. Типом для параметра `rest` всегда является массив. К теме типизированных массивов мы вернемся в главе 9. Аннотация для параметра `extraFees` сообщает компилятору, что все дополнительные аргументы должны быть числами. Код в листинге 8.11 выдает следующий результат:

```
2 args: 120
1 arg: 120
3 args: 130
6 args: 168
```

СОВЕТ

Аннотации типов для необязательных параметров применяются после вопросительного знака, например так: `discount?: number`.

Управление значениями нулевых параметров

Как объяснялось в главе 7, TypeScript по умолчанию позволяет использовать `null` и `undefined` в качестве значений для всех типов, а это означает, что функция может принимать нулевые значения для всех своих параметров, как показано в листинге 8.12.

Листинг 8.12. Передача значения null в функцию в файле index.ts

```
function calculateTax(amount: number,
    discount: number = 0, ...extraFees: number[]) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(null, 0);
console.log('Tax value: ${taxValue}');
```

Если значение `null` используется для параметра, инициализированного по умолчанию, то применяется значение по умолчанию, как если бы функция была вызвана без аргумента. Однако для обязательных параметров функция получает значение `null`, что может привести к неожиданным результатам. В приведенном примере функция `calculateTax` получает значение `null` для параметра `amount` и дает следующий результат:

```
Tax value: 0
```

Значение `null` преобразуется к числу `0` оператором умножения. Для определенных проектов это считается приемлемым, но именно такой результат приводит к неверному интерпретированию значения `null` и запутывает пользователя во время выполнения программы. Параметр компилятора `strictNullChecks` запрещает использовать `null` и `undefined` в качестве значений для всех типов, как это описано в главе 7. Она предписывает параметрам, которые могут принимать значения `null`, использовать объединение типов. Листинг 8.13 включает этот параметр компилятора.

Листинг 8.13. Изменение параметра компилятора в файле tsconfig.json из папки types

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true
  }
}
```

После сохранения конфигурационного файла запустится компилятор, который выдаст следующую ошибку, указывающую на использование аргумента `null`:

```
src/index.ts(6,29): error TS2345: Argument of type 'null' is not assignable
to parameter of type 'number'.
```

Если необходимо разрешить значения `null`, параметр может быть определен с помощью объединения типов, как показано в листинге 8.14.

Листинг 8.14. Разрешение значения null для параметра в файле index.ts из папки src

```
function calculateTax(amount: number | null, discount: number = 0,
    ...extraFees: number[]) {
  if (amount != null) {
```

```
        return (amount * 1.2) - discount
            + extraFees.reduce((total, val) => total + val, 0);
    }
}

let taxValue = calculateTax(null, 0);
console.log('Tax value: ${taxValue}');
```

Для предотвращения использования значения `null` в операторе умножения требуется защита типа. Это кажется сложным, когда вы только начинаете работать с TypeScript, но ограничение параметров, допускающих значение `null`, позволяет избежать проблем, которые в противном случае могли бы привести к неожиданным результатам. Код в листинге 8.14 выводит следующее:

```
Tax value: undefined
```

8.2.3. Результаты функции

Компилятор TypeScript попытается вывести тип результата из кода функции и автоматически будет использовать объединения типов, если функция может возвращать несколько типов. Простейший способ узнать, какой тип компилятор вывел для результата функции, — включить генерацию файлов декларации типов с помощью опции `declaration`, которая была включена в листинге 8.2. Эти файлы используются для предоставления информации о типе, когда пакет задействуется в другом проекте TypeScript (подробнее об этом — в главе 15).

Изучите содержимое файла `index.d.ts` в папке `dist`, чтобы просмотреть подробную информацию о типах, которые компилятор вывел или прочитал из аннотаций типов:

```
declare function calculateTax(amount: number | null, discount?: number,
...extraFees: number[]): number | undefined;
declare let taxValue: number | undefined;
```

Выделенная часть информации о типе функции `CalcTax` показывает тип, определенный компилятором для результата функции.

Отключение неявного возврата

В JavaScript существует необычайно мягкий подход к результатам работы функций: если для пути в коде функции не встречается оператор с ключевым словом `return`, то она автоматически возвращает значение `undefined`. Это называется *неявным возвратом*.

Зашита типа, используемая для фильтрации значений `null`, указывает на наличие в коде функции пути, который не достигает оператора `return`. Таким образом, если параметр `amount` не равен `null`, функция вернет значение типа `number`, а если равен `null` — `undefined`. В листинге 8.14 была включена опция компилятора `strictNullChecks`, поэтому компилятор сделал вывод о том, что тип результата будет `number | undefined`.

Для предотвращения неявных возвратов включите соответствующую настройку компилятора, как показано в листинге 8.15.

Листинг 8.15. Изменение конфигурации компилятора в файле tsconfig.json из папки types

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true,
    "noImplicitReturns": true
  }
}
```

Если для параметра `noImplicitReturns` установлено значение `true`, то компилятор будет сообщать об ошибке при наличии путей через функции, которые явно не возвращают результат с использованием ключевого слова `return` или не выдают ошибку. Сохраните изменения в файле `tsconfig.json`. Компилятор построит файл `index.ts` с использованием новой конфигурации и выдаст следующее:

```
src/index.ts(1,10): error TS7030: Not all code paths return a value.
```

Теперь каждый путь выполнения функции должен давать результат. Функция по-прежнему может возвращать `undefined`, но теперь это необходимо делать явно, как показано в листинге 8.16.

Листинг 8.16. Возврат результата в файле index.ts из папки src

```
function calculateTax(amount: number | null, discount: number = 0,
  ...extraFees: number[]) {
  if (amount != null) {
    return (amount * 1.2) - discount
      + extraFees.reduce((total, val) => total + val, 0);
  } else {
    return undefined;
  }
}

let taxValue = calculateTax(null, 0);
console.log('Tax value: ${taxValue}');
```

Отключение неявных возвратов гарантирует, что функции должны явно указывать выдаваемые ими результаты. Изменение в листинге 8.16 устраняет ошибку компилятора из листинга 8.14 и приводит к следующему результату:

```
Tax value: undefined
```

Использование аннотаций типов для результатов функций

Компилятор определяет тип результата функции, анализируя пути кода и создавая объединение встречающихся ему типов. Однако я предпочитаю использовать аннотацию типа для явного указания типа результата. Это позволяет мне четко

объявить ожидаемый результат функции и избежать случайного использования неправильного типа. Аннотации для результатов функции размещаются в конце сигнатуры функции, как показано в листинге 8.17.

Листинг 8.17. Аннотирование типа результата функции в файле index.ts

```
function calculateTax(amount: number, discount: number = 0,
    ...extraFees: number[]): number {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(100, 0);
console.log('Tax value: ${taxValue}');
```

В этом примере задан тип результата `number` и удален тип `null` из параметра `amount`. Явное объявление типа гарантирует, что компилятор выдаст ошибку, если мы случайно вернем из функции значение другого типа. После компиляции и выполнения код в листинге 8.17 выдаст следующий результат:

```
Tax value: 120
```

Определение функций `void`

Функции, не возвращающие результата, объявляются с использованием типа `void`, как показано в листинге 8.18.

Листинг 8.18. Определение void-функции в файле index.ts

```
function calculateTax(amount: number, discount: number = 0,
    ...extraFees: number[]): number {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

function writeValue(label: string, value: number): void {
    console.log('${label}: ${value}');
}

writeValue("Tax value", calculateTax(100, 0));
```

Функция `writeValue` не возвращает результат и имеет аннотацию типа `void`, наличие которого гарантирует, что компилятор предупредит вас, если будет использовано ключевое слово `return` или если функция будет присваивать значения.

ПРИМЕЧАНИЕ

Тип `never` может быть использован в качестве типа результата для функций, которые никогда не завершатся, например для тех, которые всегда выбрасывают исключение.

Код в листинге 8.18 выдает следующий результат:

```
Tax value: 120
```

8.2.4. Перегрузка типов функций

Объединения типов позволяют определить диапазон типов для параметров и результатов функции, но не позволяют точно выразить отношения между ними (листинг 8.19).

Листинг 8.19. Определение функции с объединениями типов в файле index.ts

```
function calculateTax(amount: number | null): number | null {
    if (amount != null) {
        return amount * 1.2;
    }
    return null;
}

function writeValue(label: string, value: number): void {
    console.log(`${label}: ${value}`);
}

let taxAmount: number | null = calculateTax(100);
if (typeof taxAmount === "number") {
    writeValue("Tax value", taxAmount);
}
```

Аннотация типа в листинге 8.19 описывает типы, которые принимает функция `calculateTax`, сообщая пользователям, что функция принимает либо `number`, либо `null` и возвращает `number` или `null`. Информация, предоставленная объединением типов, корректна, но не полностью описывает ситуацию. Она не указывает связи между параметром и типами результатов: если на вход функции передается параметр типа `number`, то и результатом будет `number`, а если передается значение `null`, то и результат равен `null`. Отсутствие деталей в типах функции означает, что пользователь функции должен использовать защиту типа результата для удаления значений `null`, несмотря на то что значение `100` является типом `number` и всегда будет давать числовой результат.

Для описания отношений между типами, используемыми функцией, TypeScript поддерживает перегрузки типов, как показано в листинге 8.20.

ПРИМЕЧАНИЕ

Следует отметить, что это не такая перегрузка функций, как в C# и Java. Здесь перегружается только информация о типе с целью проверки типа. Как видно из листинга 8.20, существует только одна реализация функции, которая тем не менее отвечает за работу со всеми типами, используемыми в перегрузках.

Листинг 8.20. Перегрузка типов функций в файле index.ts

```
function calculateTax(amount: number): number;
function calculateTax(amount: null): null;
function calculateTax(amount: number | null): number | null {
    if (amount != null) {
        return amount * 1.2;
    }
    return null;
}
```

```
function writeValue(label: string, value: number): void {
    console.log(`${label}: ${value}`);
}

let taxAmount: number = calculateTax(100);
//if (typeof taxAmount === "number") {
//    writeValue("Tax value", taxAmount);
//}
```

Каждая перегрузка типа определяет комбинацию типов, поддерживаемых функцией, и описывает сопоставление между параметрами и результатом, который они дают, как показано на рис. 8.4.

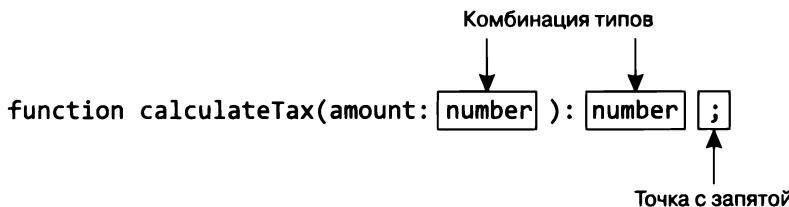


Рис. 8.4. Перегрузка типа функции

Перегрузки типов заменяют определение функции в качестве информации о типе, используемой компилятором TypeScript. Это означает, что можно использовать только определенные комбинации типов. При вызове функции компилятор может определить тип результата на основе типа предоставленных аргументов, что позволяет определить переменную `taxAmount` как `number` и избавляется от необходимости использовать защиту типа для передачи результата в функцию `writeValue`. Компилятор знает, что `taxAmount` может быть только числом, и не требует сужения типа. Код в листинге 8.20 при компиляции и выполнении выдает следующий результат:

Tax value: 120

СОВЕТ

Вы также можете выразить связь между параметрами и результатами с помощью функций условных типов, которая описана в главе 13.

8.2.5. Функции утверждения

Функция утверждения (assert function) оценивает условие выражения и, как правило, выдает ошибку, если результат не является `true`. В чистом JavaScript, где статические типы TypeScript недоступны, подобные функции иногда используются в качестве защиты типов. Однако у функций утверждения есть проблема: компилятор TypeScript не может определить их влияние на типы (листинг 8.21).

Листинг 8.21. Использование функции утверждения в файле index.ts

```
function check(expression: boolean) {
    if (!expression) {
        throw new Error("Expression is false");
    }
}

function calculateTax(amount: number | null): number {
    check(typeof amount == "number");
    return amount * 1.2;
}

let taxAmount: number = calculateTax(100);
console.log('Tax value: ${taxAmount}');
```

Функция `check` принимает параметр типа `boolean` и выдает ошибку, если ей передается `false`. Это базовый шаблон функции утверждения.

Функция `calculateTax` принимает аргумент `number | null` с помощью функции `check` сужает тип таким образом, чтобы значения `null` вызывали ошибки, а для получения результата использовались значения типа `number`.

Проблема этого кода в том, что компилятор TypeScript не понимает, что функция `check` гарантирует обработку только значений типа `number`. При компиляции кода выводится следующее сообщение:

```
src/index.ts(9,12): error TS18047: 'amount' is possibly 'null'.
```

Ключевое слово `asserts` может быть использовано для обозначения функции утверждения (`assert function`), которая позволяет компилятору TypeScript учитывать эту функцию, как показано в листинге 8.22.

Листинг 8.22. Обозначение функции утверждения в файле index.ts

```
function check(expression: boolean) : asserts expression {
    if (!expression) {
        throw new Error("Expression is false");
    }
}

function calculateTax(amount: number | null): number {
    check(typeof amount == "number");
    return amount * 1.2;
}

let taxAmount: number = calculateTax(100);
console.log('Tax value: ${taxAmount}');
```

Ключевое слово `asserts` используется в качестве типа результата, за которым следует имя параметра, утверждаемого функцией, как показано на рис. 8.5.

Компилятор TypeScript может учитывать эффект функции `check` и знает, что функция `calculateTax` сужает тип параметра `amount`, чтобы исключить значения `null`.

Существует вариант функций утверждения, которые оперируют непосредственно типами, а не просто оценивают выражение, как показано в листинге 8.23.

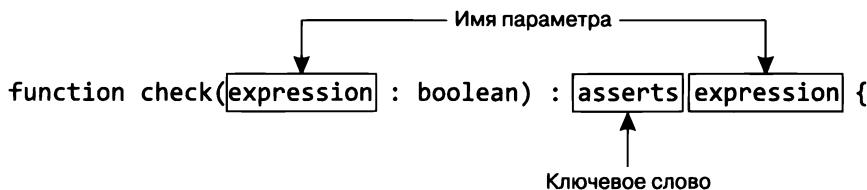


Рис. 8.5. Обозначение функции утверждения

Листинг 8.23. Сужение типов непосредственно в файле index.ts

```

function checkNumber(val: any): asserts val is number {
    if (typeof val != "number") {
        throw new Error("Not a number");
    }
}

function calculateTax(amount: number | null): number {
    checkNumber(amount);
    return amount * 1.2;
}

let taxAmount: number = calculateTax(100);
console.log('Tax value: ${taxAmount}');
    
```

В данном примере за ключевым словом `asserts` следует `val is number`, что сообщает компилятору TypeScript, что эффект функции `checkNumber` заключается в том, что она гарантирует, что параметр `val` является значением типа `number`.

РЕЗЮМЕ

В этой главе были показаны возможности, которые TypeScript предоставляет для работы с функциями. Мы поговорили о методах предотвращения дублирования определений функций, рассмотрели различные способы описания параметров и результатов функции, а также разобрали, как переопределять типы функций для создания более точных соответствий между типами параметров и получаемыми результатами.

- Функции определяются с помощью стандартного синтаксиса JavaScript, но могут быть аннотированы статическими типами для параметров и результата.
- Необязательные параметры обозначаются символом `?` и могут быть опущены при вызове функции.
- Параметры по умолчанию определяются путем присвоения значения при определении функции.
- Параметры `rest` обозначаются многоточием и используются для работы с произвольным числом параметров.

- Компилятор TypeScript можно настроить так, чтобы значения типа `null` и `undefined` можно было использовать только для параметров или результатов, тип которых включает эти значения в объединение.
- Компилятор TypeScript можно настроить так, чтобы функции явно возвращали результаты, если они определяют тип результата.
- JavaScript не поддерживает перегрузку функций, но с помощью TypeScript можно определить перегрузки типов, которые описывают определенные комбинации типов параметров и получаемых ими типов результатов.
- Функции `Assert` могут применяться для предоставления информации о типе компилятору TypeScript, подобно тому как для типов JavaScript используется защита типов.

В следующей главе будет рассмотрено, как TypeScript обращается с простыми структурами данных.

9

Массивы, кортежи и перечисления

В этой главе

- ✓ Ограничение типов, которые могут содержаться в массиве.
- ✓ Создание массивов фиксированной длины с помощью кортежей.
- ✓ Использование перечислений для группировки связанных значений.
- ✓ Применение литеральных типов значений для определения фиксированного набора допустимых значений.
- ✓ Создание псевдонима типа для упрощения работы со сложными определениями типов.

До сих пор в примерах к этой части книги внимание было сосредоточено на примитивных типах, что позволило продемонстрировать основные возможности TypeScript. В реальных же проектах связанные свойства данных группируются для создания объектов. В этой главе описана поддержка TypeScript для простых структур данных, начиная с массивов. В табл. 9.1 приведено краткое содержание главы.

В качестве краткой справки в табл. 9.2 перечислены параметры компилятора TypeScript, используемые в данной главе.

Таблица 9.1. Краткое содержание главы

Задача	Решение	Листинг
Ограничить диапазон типов, которые массив может содержать	Примените аннотацию типа или разрешите компилятору выводить типы из используемого значения для инициализации массива	4–9
Определить массив фиксированной длины с указанными типами для каждого значения	Используйте кортежи	10–14
Определить массив переменной длины с указанными типами для каждого значения	Используйте кортежи с rest-элементом	15
Сослаться на коллекцию связанных значений через одно имя	Используйте перечисления	16–25
Определить тип, который может быть назначен только конкретным значениям	Используйте литеральные типы значений	26–32
Избежать дублирования при описании сложного типа	Используйте псевдоним типа	33

Таблица 9.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода
outDir	Задает каталог, в котором будут размещены файлы JavaScript
rootDir	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
declaration	Создает файлы деклараций типов, которые могут быть полезны для понимания того, как выводятся типы. Более подробно эти файлы описаны в главе 15
strictNullChecks	Не допускает присвоение null и undefined в качестве значений для других типов

9.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В этой главе мы продолжим использовать проект `types`, созданный в главе 7. Для начала замените содержимое файла `index.ts` в папке `src` кодом, показанным в листинге 9.1.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 9.1. Содержимое файла index.ts из папки src

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: ${price.toFixed(2)}`);
}

let hatPrice = 100;
let glovesPrice = 75;
let umbrellaPrice = 42;

writePrice("Hat", calculateTax(hatPrice));
writePrice("Gloves", calculateTax(glovesPrice));
writePrice("Umbrella", calculateTax(umbrellaPrice));
```

Закомментируйте параметры компилятора, как показано в листинге 9.2, чтобы сбросить конфигурацию компилятора.

Листинг 9.2. Отключение параметров компилятора в файле tsconfig.json из папки types

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    // "strictNullChecks": true,
    // "noImplicitReturns": true
  }
}
```

Откройте новое окно командной строки, перейдите в папку types и выполните команду, показанную в листинге 9.3, для запуска компилятора TypeScript, чтобы скомпилированный код выполнялся автоматически.

Листинг 9.3. Запуск компилятора TypeScript

```
npm start
```

Компилятор скомпилирует код в файле index.ts, выполнит вывод, а затем перейдет в режим наблюдения, выдав следующий результат:

```
6:58:20 AM - File change detected. Starting incremental compilation...
6:58:21 AM - Found 0 errors. Watching for file changes.
Price for Hat: $120.00
Price for Gloves: $90.00
Price for Umbrella: $50.40
```

9.2. РАБОТА С МАССИВАМИ

Как объяснялось в главе 8, массивы JavaScript могут содержать различные типы данных и иметь переменную длину, что позволяет добавлять и удалять значения без необходимости явного изменения размера массива. TypeScript не ограничивает гибкость размеров массивов, но предоставляет возможность ограничивать типы данных, которые они содержат, с помощью аннотаций типов, как показано в листинге 9.4.

Листинг 9.4. Использование массивов в файле index.ts из папки src

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: $$${price.toFixed(2)}');
}

let prices: number[] = [100, 75, 42];
let names: string[] = ["Hat", "Gloves", "Umbrella"];

writePrice(names[0], calculateTax(prices[0]));
writePrice(names[1], calculateTax(prices[1]));
writePrice(names[2], calculateTax(prices[2]));
```

Тип массива задается квадратными скобками, помещаемыми после имени типа в аннотации, как показано на рис. 9.1.

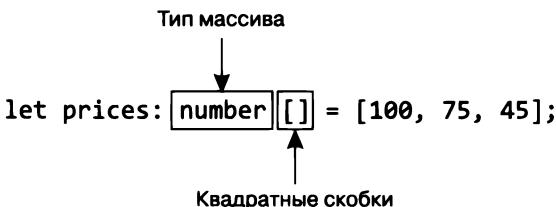


Рис. 9.1. Аннотация типа массива

TypeScript использует аннотацию для ограничения операций, которые можно выполнять с массивом, указанным типом: один из массивов в листинге ограничен числовыми значениями, а другой — строковыми. В листинге 9.5 использован метод JavaScript `forEach` для массивов. Обратите внимание, что функция, используемая для обработки значений массива, типизирована в соответствии с типами массивов.

Листинг 9.5. Выполнение операций над типизированными массивами в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}
```

```
function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: $$${price.toFixed(2)}');
}

let prices: number[] = [100, 75, 42];
let names: string[] = ["Hat", "Gloves", "Umbrella"];

prices.forEach((price: number, index: number) => {
    writePrice(names[index], calculateTax(price));
});
```

СОВЕТ

Вы можете использовать круглые скобки при описании массива, содержащего несколько типов, например, при использовании объединения типов (см. главу 8) или пересечения типов (см. главу 10). Например, массив, элементами которого могут быть числовые или строковые значения, может быть аннотирован как `(number | string) []`, при этом круглые скобки вокруг объединения типов не позволят компилятору предположить, что объединение происходит между одним числом или массивом строк.

Первый аргумент функции, передаваемый методу `forEach`, принимает числовое значение, поскольку это соответствует типу обрабатываемого массива. TypeScript гарантирует, что функция будет выполнять только те операции, которые разрешены для числовых значений. Код в листинге 9.5 при компиляции и выполнении выдает следующий результат:

```
Price for Hat: $120.00
Price for Gloves: $90.00
Price for Umbrella: $50.40
```

9.2.1. Вывод типов массивов

В листинге 9.5 мы использовали аннотации типов, чтобы явно указать, что массивы типизированы. Однако компилятор TypeScript способен автоматически выводить типы, и тот же пример можно выразить без аннотаций типов (листинг 9.6).

Листинг 9.6. Автоматический вывод типов в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: $$${price.toFixed(2)}');
}

let prices = [100, 75, 42];
let names = ["Hat", "Gloves", "Umbrella"];

prices.forEach((price, index) => {
    writePrice(names[index], calculateTax(price));
});
```

Компилятор может определять типы массивов на основе набора значений, присваиваемых при инициализации массивов, и использует эти выведенные типы при выполнении метода `forEach`.

Компилятор умеет выводить типы, но если вы не получаете ожидаемых результатов, можете просмотреть файлы, которые генерирует компилятор при включенной опции `declaration`. Эта опция генерирует файлы деклараций типов, которые используются для предоставления информации о типах при использовании пакета в другом проекте TypeScript. Подробнее о них говорится в главе 15.

Вот типы, которые компилятор определил для массивов в листинге 9.6, содержащихся в файле `index.d.ts` в папке `dist`:

```
...
declare let prices: number[];
declare let names: string[];
...
```

О ключевом слове `declare` рассказано в главе 15. Пока же достаточно убедиться, что компилятор правильно определил типы массивов по начальным значениям.

9.2.2. Проблемы при выводе типов массивов

Компилятор определяет типы массивов на основе значений, которые используются при создании массива. Это может привести к ошибкам с типами, если значения, используемые для заполнения массива, случайно смешиваются, как показано в листинге 9.7.

Листинг 9.7. Смешение типов массивов в файле `index.ts`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: ${price.toFixed(2)}`);
}

let prices = [100, 75, 42, "20"];
let names = ["Hat", "Gloves", "Umbrella", "Sunglasses"];

prices.forEach((price, index) => {
    writePrice(names[index], calculateTax(price));
});
```

Использование нового значения, которое имеет другой тип, для инициализации массива `price` приводит к следующей ошибке при компиляции кода:

```
src/index.ts(13,43): error TS2345: Argument of type 'string | number' is
not assignable to parameter of type 'number'.
  Type 'string' is not assignable to type 'number'.
```

Если взглянуть на файл `index.d.ts` в папке `dist`, то можно увидеть, что компилятор TypeScript вывел наименьший набор типов, который может описать значения, используемые для инициализации массива:

```
declare let prices: (string | number)[];
```

Изменение типа массива приводит к появлению сообщения об ошибке, поскольку функция, переданная в метод `forEach`, обрабатывает значения типа `number`, в то время как теперь они являются частью объединения `string | number`. На простом примере легко увидеть причину проблемы, но все усложняется, когда начальные значения для массива поступают из разных частей приложения. Рекомендую явно объявлять тип массива, чтобы избегать проблем, которые приводят к ошибке компилятора при попытке добавить строку в числовой массив.

9.2.3. Проблемы с пустыми массивами

Другая причина использования аннотаций типов для массивов заключается в том, что компилятор будет выводить тип `any` для массивов, которые создаются пустыми (листинг 9.8).

Листинг 9.8. Создание пустого массива в файле `index.ts`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: ${price.toFixed(2)}`);
}

let prices = [];
prices.push(...[100, 75, 42, "20"]);
let names = ["Hat", "Gloves", "Umbrella", "Sunglasses"];

prices.forEach((price, index) => {
    writePrice(names[index], calculateTax(price));
});
```

В нашем случае компилятору не предоставляется информация о типе для массива `prices`, так как он создан без начальных значений. Единственный вариант, доступный компилятору, — использовать тип `any`, поскольку у него нет другой информации для работы. В этом вы можете убедиться, изучив файл `index.d.ts` в папке `dist`.

```
declare let prices: any[];
```

Несмотря на то что в массив добавляются значения разных типов (как `number`, так и `string`), код в листинге 9.8 компилируется без ошибок и выдает следующие результаты:

```
Price for Hat: $120.00
Price for Gloves: $90.00
Price for Umbrella: $50.40
Price for Sunglasses: $24.00
```

Разрешение компилятору определить тип пустого массива приведет к созданию пробела в процессе проверки типов. Код работает, потому что оператор умножения JavaScript автоматически приводит строковые значения к числовым. В редких случаях это может быть полезно, но, скорее всего, такая практика будет иметь случайный характер, и именно по этой причине следует использовать явные типы.

Подводные камни типа never

TypeScript по-разному выводит типы для пустых массивов, когда значения `null` и `undefined` не могут быть назначены другим типам. Чтобы увидеть разницу, измените конфигурацию компилятора, как показано в листинге 9.9.

Листинг 9.9. Настройка компилятора в файле `tsconfig.json` из папки `types`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true,
  }
}
```

Параметр `strictNullChecks` указывает компилятору на ограничение использования значений `null` и `undefined` и не позволяет ему использовать их при выводе типа пустого массива. Вместо этого компилятор выводит тип `never`, а это означает, что в массив ничего не может быть добавлено. При компиляции и выполнении кода, приведенного в листинге 9.9, выдается следующая ошибка:

```
src/index.ts(10,13): error TS2345: Argument of type 'string | number' is
not assignable to parameter of type 'never'.
  Type 'string' is not assignable to type 'never'.
```

Вывод типа `never` гарантирует, что массив не сможет обойти проверку типов и код не будет скомпилирован до тех пор, пока для массива не будет указан тип или массив не будет инициализирован с помощью значений, позволяющих компилятору вывести менее ограничительный тип.

9.3. КОРТЕЖИ

Базовые кортежи — это массивы фиксированной длины, где каждый элемент имеет разный тип. Кортежи — это структура данных, предоставляемая компилятором TypeScript и реализуемая с помощью стандартных массивов JavaScript. В листинге 9.10 демонстрируется, как определяются и используются кортежи. (Существует также более сложный тип кортежей, о котором мы поговорим позднее.)

Листинг 9.10. Использование кортежей в файле index.ts из папки src

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: ${price.toFixed(2)}`);
}

let hat: [string, number] = ["Hat", 100];
let gloves: [string, number] = ["Gloves", 75];

writePrice(hat[0], hat[1]);
writePrice(gloves[0], gloves[1]);
```

Кортежи определяются с помощью квадратных скобок, в которых указываются типы для каждого элемента, разделенные запятыми, как показано на рис. 9.2.



Рис. 9.2. Определение кортежа

Тип кортежа `hat` в листинге 9.10 — `[string, number]`, то есть это кортеж с двумя элементами, где первый элемент — строка, а второй — число. Доступ к элементам кортежа осуществляется с использованием синтаксиса индекса массива, так что первым элементом кортежа `hat` будет `hat[0]`.

Результат выполнения кода из листинга 9.10:

```
Price for Hat: $100.00
Price for Gloves: $75.00
```

Кортежи должны быть определены с помощью аннотаций типа, в противном случае компилятор будет рассматривать их как обычный массив с типом, представляющим собой объединение всех используемых при инициализации значений. Например, без аннотации типа, как показано на рис. 9.2, компилятор предположил бы, что тип значения, присвоенного переменной `hat`, равен `[string | number]`, который будет обозначать массив переменной длины, где каждый элемент может быть как строкой, так и числом.

9.3.1. Обработка кортежей

Ограничения на количество элементов и типы элементов полностью накладываются компилятором TypeScript. Во время выполнения кортеж реализуется как обычный массив JavaScript. Это означает, что с кортежами можно работать так же, как с обычными массивами JavaScript (листинг 9.11).

Листинг 9.11. Обработка элементов в кортеже в файле index.ts из папки src

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: $$${price.toFixed(2)}');
}

let hat: [string, number] = ["Hat", 100];
let gloves: [string, number] = ["Gloves", 75];

hat.forEach((h: string | number) => {
    if (typeof h === "string") {
        console.log('String: ${h}');
    } else {
        console.log('Number: ${h.toFixed(2)})');
    }
});
```

Чтобы обработать все значения кортежа, функция, передаваемая методу `forEach`, должна получить значения `string | number`, которые затем сужаются с помощью защиты типа. В нашем примере для наглядности используются аннотации типов, но компилятор правильно определит тип объединения на основе типов элементов кортежа. Результат выполнения кода из листинга 9.11:

```
String: Hat
Number: 100.00
```

Поскольку кортежи являются массивами, они могут быть деструктурированы для доступа к отдельным значениям, что в целом упрощает работу с кортежами, как показано в листинге 9.12.

Листинг 9.12. Деструктуризация кортежей в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: $$${price.toFixed(2)}');
}

let hat: [string, number] = ["Hat", 100];
let gloves: [string, number] = ["Gloves", 75];

let [hatname, hatprice] = hat;
console.log('Name: ${hatname}');
console.log('Price: ${hatprice.toFixed(2)})');
```

Кортеж `hat` деструктурируется, его значения присваиваются переменным `hatname` и `hatprice`, которые выводятся в консоль. В примере изменился не вывод, а только способ доступа к значениям кортежей.

9.3.2. Использование типов кортежей

Кортежи имеют отдельный тип, который можно использовать так же, как и любой другой. Это означает возможность создавать массивы кортежей, включать их в объединения типов и использовать защиту типов для сужения значений до определенных типов кортежей (листинг 9.13).

Листинг 9.13. Использование кортежей в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: $$${price.toFixed(2)}');
}

let hat: [string, number] = ["Hat", 100];
let gloves: [string, number] = ["Gloves", 75];

let products: [string, number][] = [["Hat", 100], ["Gloves", 75]];
let tupleUnion: ([string, number] | boolean)[]
    = [true, false, hat, ...products];

tupleUnion.forEach((elem: [string, number] | boolean) => {
    if (elem instanceof Array) {
        let [str, num] = elem;
        console.log('Name: ${str}');
        console.log('Price: ${num.toFixed(2)})');
    } else if (typeof elem === "boolean") {
        console.log('Boolean Value: ${elem}');
    }
});
```

Обилие квадратных скобок может сбить с толку, и для правильного описания комбинации типов может потребоваться несколько попыток, но этот пример иллюстрирует, как тип кортежа может использоваться так же, как и любой другой тип, хотя и с одним важным отличием от предыдущих примеров в этой части книги: для определения того, является ли значение кортежем, нельзя использовать ключевое слово `typeof` в листинге 9.13. Кортежи реализуются с помощью стандартных массивов JavaScript, а для проверки их типов требуется ключевое слово `instanceof`, о котором говорилось в главе 4. При компиляции и выполнении код в листинге 9.13 выдает следующий результат:

```
Boolean Value: true
Boolean Value: false
String Value: Hat
Number Value: 100
String Value: Hat
Number Value: 100
String Value: Gloves
Number Value: 75
```

9.3.3. Кортежи с необязательными элементами

Кортежи могут содержать необязательные элементы, которые обозначаются символом ?. Кортеж по-прежнему остается структурой фиксированной длины, а необязательный элемент будет иметь значение `undefined`, если оно не определено (листинг 9.14).

Листинг 9.14. Использование необязательного элемента в файле `index.ts`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: $$${price.toFixed(2)}');
}

let hat: [string, number, number?] = ["Hat", 100];
let gloves: [string, number, number?] = ["Gloves", 75, 10];

[hat, gloves].forEach(tuple => {
    let [name, price, taxRate] = tuple;
    if (taxRate != undefined) {
        price += price * (taxRate / 100);
    }
    writePrice(name, price);
});
```

В данном примере тип кортежа содержит необязательный элемент типа `number`. (Помните, что в кортеже могут иметься несколько необязательных элементов, но они должны быть последними элементами в определении типа кортежа.)

Тип необязательного элемента представляет собой объединение указанного типа с типом `undefined`, что в нашем примере приводит к типу `number | undefined`. Значение элемента будет `undefined`, если значение не было предоставлено, и код, обрабатывающий кортеж, обязан сузить тип, исключив значения типа `undefined`.

Определение необязательного элемента означает, что компилятор TypeScript не обращает внимания на отсутствие соответствующего значения, как, например, в этом случае:

```
...
let hat: [string, number, number?] = ["Hat", 100];
...
```

Значение для третьего элемента кортежа отсутствует, но компилятор обрабатывает код без проблем и выдает следующий результат:

```
Price for Hat: $100.00
Price for Gloves: $82.50
```

9.3.4. Определение кортежей с rest-элементами

Кортежи могут также включать в себя элемент `rest` (...), который можно использовать для сопоставления нескольких значений заданного типа. Это дает возможность создавать кортежи переменной длины, которые не имеют жестко определенной структуры базовых кортежей. В листинге 9.15 приведен пример использования кортежа с элементом `rest`.

Листинг 9.15. Использование элемента `rest` в файле `index.ts`

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: ${price.toFixed(2)}`);
}

let hat: [string, number, number?, ...number[]} = ["Hat", 100, 10, 1.20, 3, 0.95];
let gloves: [string, number, number?, ...number[]} = ["Gloves", 75, 10];

[hat, gloves].forEach(tuple => {
    let [name, price, taxRate, ...coupons] = tuple;
    if (taxRate != undefined) {
        price += price * (taxRate / 100);
    }
    coupons.forEach(c => price -= c);
    writePrice(name, price);
});
```

В данном примере элемент `rest` кортежа деструктурируется в массив с именем `coupons`, который затем обрабатывается в цикле `forEach`, что дает следующий результат:

```
Price for Hat: $104.85
Price for Gloves: $82.50
```

Эта возможность мне не нравится, потому что переменная длина, вносимая элементами `rest`, подрывает фиксированную структуру, которая делает кортежи полезными. Я редко использую эту возможность, применяя ее только в описании кода JavaScript (см. главу 15).

9.4. ПЕРЕЧИСЛЕНИЯ

Перечисление позволяет обращаться к коллекции значений по имени, что упрощает чтение кода и обеспечивает последовательное использование фиксированного набора значений. Подобно кортежам, перечисления являются функцией, предоставляемой компилятором TypeScript. В листинге 9.16 показано определение и использование перечисления.

Листинг 9.16. Использование перечисления в файле index.ts

```

function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: ${price.toFixed(2)}`);
}

enum Product { Hat, Gloves, Umbrella }

let products: [Product, number][] =
    [[Product.Hat, 100], [Product.Gloves, 75]];

products.forEach((prod: [Product, number]) => {
    switch (prod[0]) {
        case Product.Hat:
            writePrice("Hat", calculateTax(prod[1]));
            break;
        case Product.Gloves:
            writePrice("Gloves", calculateTax(prod[1]));
            break;
        case Product.Umbrella:
            writePrice("Umbrella", calculateTax(prod[1]));
            break;
    }
});

```

Перечисление определяется с помощью ключевого слова `enum`, за которым следует имя, а затем список значений в фигурных скобках, как показано на рис. 9.3.

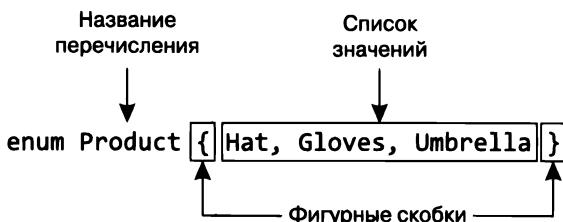


Рис. 9.3. Определение перечисления

Доступ к значениям перечислений осуществляется в форме `<enum>.〈value〉`. Например, к значению `Hat`, определяемому перечислением `Product`, можно обратиться как `Product.Hat`:

```

...
case Product.Hat:
...

```

Перечисление используется так же, как и любой другой тип. В приведенном примере показано применение перечисления `Product` в кортеже и операторе `switch`. Код в листинге 9.16 при компиляции и выполнении выдает следующий результат:

```

Price for Hat: $120.00
Price for Gloves: $90.00

```

9.4.1. Принцип работы перечислений

Перечисления полностью реализуются компилятором TypeScript, опираясь на проверку типов при компиляции и стандартные возможности JavaScript во время выполнения. Каждому значению перечисления автоматически присваивается соответствующее числовое значение, начиная с нуля в качестве значения по умолчанию. Это означает, что для имен `Hat`, `Gloves` и `Umbrella` в перечислении `Product` используются числа 0, 1 и 2, как показано в листинге 9.17.

Листинг 9.17. Использование значения перечисления в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: ${price.toFixed(2)}');
}

enum Product { Hat, Gloves, Umbrella }
[Product.Hat, Product.Gloves, Product.Umbrella].forEach(val => {
    console.log('Number value: ${val}');
});
```

Выделенные операторы передают в `console.log` каждое значение из перечисления `Product`. Каждое значение перечисления является числом, и код в листинге 9.17 выдает следующий результат:

```
Number value: 0
Number value: 1
Number value: 2
```

Поскольку перечисления основаны на числовых значениях JavaScript, им может быть присвоено значение типа `number`, которое отображается в виде числового значения, как показано в листинге 9.18.

Листинг 9.18. Использование перечислений и числовых значений в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: ${price.toFixed(2)}');
}

enum Product { Hat, Gloves, Umbrella }

let productValue: Product = 0;
let productName: string = Product[productValue];
console.log('Value: ${productValue}, Name: ${productName}');
```

Компилятор обеспечивает проверку типов перечислений, поэтому при попытке сравнить значения из разных перечислений, даже если они имеют одинаковое значение `number`, возникнет ошибка. Перечисления предоставляют синтаксис

в стиле индексатора массива, который можно использовать для получения имени значения, например, так:

```
...
let productName: string = Product[productValue];
...
```

Результатом этой операции является строка, содержащая имя значения перечисления, которым в данном примере является `Hat`. Код в листинге 9.18 выдает следующий результат:

```
Value: 0, Name: Hat
```

Использование специфических значений перечислений

По умолчанию компилятор TypeScript присваивает значения перечислению, используя тип `number`, начиная с нуля. Он вычисляет значения, инкрементируя предыдущее. В случае с перечислением `Product` из листинга 9.18 компилятор присваивает `0` значению `Hat`, `1` — `Gloves` и `2` — `Umbrella`. Если вы хотите узнать, какие значения были присвоены перечислению, то можно изучить содержимое файлов декларации типов, которые генерируются компилятором, когда параметр `declarations` установлен в `true`. Открыв файл `index.d.ts` в папке `dist`, вы увидите присвоенные компилятором значения для перечисления `Product`:

```
...
declare enum Product {
    Hat = 0,
    Gloves = 1,
    Umbrella = 2
}
...
```

Перечисления также могут быть определены с помощью литеральных значений, если требуется конкретное значение, как показано в листинге 9.19. Это удобно, когда перечисление представляет собой реальный набор значений.

Листинг 9.19. Использование постоянного значения перечисления в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: ${price.toFixed(2)}`);
}

enum Product { Hat, Gloves = 20, Umbrella }

let productValue: Product = 0;
let productName: string = Product[productValue];
console.log('Value: ${productValue}, Name: ${productName}'');
```

Я присвоил `Gloves` значение `20`. Компилятор все равно сгенерирует остальные значения, необходимые для перечисления, и изучение файла `index.d.ts` покажет, что компилятор вычислил значения для `Hat` и `Umbrella`.

```
...
declare enum Product {
    Hat = 0,
    Gloves = 20,
    Umbrella = 21
}
...
```

Для генерации значений перечисления компилятором используется предыдущее значение, независимо от того, было ли оно выбрано программистом или сгенерировано автоматически. Для перечисления, приведенного в листинге 9.19, компилятор применил значение, присвоенное `Gloves`, для генерации значения `Umbrella`. Код в листинге 9.19 выдает следующий результат:

```
Value: 0, Name: Hat
```

ВНИМАНИЕ

Компилятор обращается к предыдущему значению только при генерации числового значения и не проверяет, не было ли это значение уже использовано, что может привести к дублированию значений в перечислении.

Компилятор оценивает простые выражения для значений перечисления, как показано в листинге 9.20. Это означает, что значения могут зависеть от других значений того же или другого перечисления, а также совсем иного значения.

Листинг 9.20. Использование выражений в перечислении в файле index.ts из папки src

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: ${price.toFixed(2)}');
}

enum OtherEnum { First = 10, Two = 20 }
enum Product { Hat = OtherEnum.First + 1, Gloves = 20,
    Umbrella = Hat + Gloves }

let productValue: Product = 11;
let productName: string = Product[productValue];
console.log('Value: ${productValue}, Name: ${productName}'');
```

Значение `Hat` присваиваетсяся с помощью выражения, использующего значение `OtherEnum` и оператор сложения, а значение `Umbrella` вычисляется как сумма значений `Hat` и `Gloves`. Просмотр файла `index.d.ts` в папке `dist` показывает, что компилятор оценил выражения для определения значений перечисления `Product`.

```
...
declare enum Product {
    Hat = 11,
    Gloves = 20,
    Umbrella = 31
}
...
```

Эти возможности могут быть полезны, однако необходимо внимательно следить за тем, чтобы случайно не создать дублирующие значения или не получить неожиданные результаты. Мой совет — не усложняйте перечисления и по возможности предоставьте компилятору генерацию чисел. Код в листинге 9.20 выдает следующий результат:

```
Value: 11, Name: Hat
```

9.4.2. Строковые перечисления

Реализация перечислений по умолчанию представляет каждое значение числом, но компилятор может также использовать для перечислений значения типа `string`, как показано в листинге 9.21.

СОВЕТ

Перечисление может содержать как строковые, так и числовые значения, хотя эта возможность широко не используется.

Листинг 9.21. Использование строкового перечисления в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log('Price for ${product}: ${price.toFixed(2)}');
}

enum OtherEnum { First = 10, Two = 20 }
enum Product { Hat = OtherEnum.First + 1, Gloves = 20,
    Umbrella = Hat + Gloves }

let productValue: Product = 11;
let productName: string = Product[productValue];
console.log('Value: ${productValue}, Name: ${productName}');

enum City { London = "London", Paris = "Paris", NY = "New York" }
console.log('City: ${City.London}'');
```

Для каждого имени значения перечисления должно быть предоставлено строковое значение, но преимущество использования строковых значений заключается в том, что их легче распознать при отладке или в лог-файлах, как это показано в листинге 9.21:

```
Value: 11, Name: Hat
City: London
```

9.4.3. Ограничения перечислений

Перечисления довольно полезны, однако у них есть некоторые ограничения, поскольку они представляют собой функцию, которая полностью реализуется компилятором TypeScript и затем транслируется в чистый JavaScript.

Ограничения проверки значений

Хотя компилятор хорошо справляется с проверкой типов для перечислений, он не предпринимает никаких дополнительных шагов, чтобы гарантировать, что используются только допустимые числовые значения. В листинге 9.21 были выбраны конкретные значения для некоторых членов перечисления `Product`, а это означает, что данное утверждение является проблемным:

```
...
let productValue: Product = 0;
...
```

Компилятор не предотвращает присвоение числового значения переменной, тип которой является перечислением, даже если число не соответствует ни одному из значений перечисления. Поэтому листинг 9.21 содержит вывод `undefined`, так как не удается найти соответствующее имя `Product` для числового значения. Аналогичная проблема возникает, когда функция использует в качестве типа результата перечисление, поскольку компилятор позволит ей вернуть любое числовое значение.

СОВЕТ

Для строковых перечислений эта проблема не возникает, поскольку они реализуются иначе и могут присваивать значения только из перечисления.

Ограничения защиты типа

Связанная с этим проблема возникает при использовании защиты типов. Проверка типов осуществляется с помощью ключевого слова JavaScript `typeof`, а поскольку перечисления реализуются с помощью числовых значений JavaScript, то `typeof` не в состоянии различить значения `enum` и `number` (листинг 9.22).

Листинг 9.22. Использование защиты типа в файле index.ts

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: ${price.toFixed(2)}`);
}

enum OtherEnum { First = 10, Two = 20 }
enum Product { Hat = OtherEnum.First + 1, Gloves = 20,
    Umbrella = Hat + Gloves }

let productValue: Product = Product.Hat;
if (typeof productValue === "number") {
    console.log("Value is a number");
}

let unionValue: number | Product = Product.Hat;
if (typeof unionValue === "number") {
    console.log("Value is a number");
}
```

Код, приведенный в листинге 9.22, при компиляции и выполнении выдает следующий результат:

```
Value is a number
Value is a number
```

Константные перечисления

Компилятор TypeScript создает объект, который реализует перечисление. В некоторых приложениях использование этого объекта может оказаться на производительности, и вместо него лучше воспользоваться другим подходом.

СОВЕТ

Данный подход редко требуется в большинстве проектов.

Чтобы продемонстрировать, как компилятор использует объект для реализации перечисления, в листинге 9.23 код в файле `index.ts` упрощен таким образом, что он определяет перечисление и содержит оператор, присваивающий переменной значение перечисления.

Листинг 9.23. Упрощение кода в файле index.ts из папки src

```
enum Product { Hat, Gloves, Umbrella }
let productValue = Product.Hat;
```

Открыв файл `index.js` в папке `dist`, вы увидите следующий код:

```
...
var Product;
(function (Product) {
    Product[Product["Hat"] = 0] = "Hat";
    Product[Product["Gloves"] = 1] = "Gloves";
    Product[Product["Umbrella"] = 2] = "Umbrella";
})(Product || (Product = {}));
let productValue = Product.Hat;
...
```

Не обязательно понимать, как работает этот код. Важно, что здесь создается объект `Product`, который используется при присвоении значения переменной `productValue`.

Для того чтобы компилятор не использовал объект для реализации перечисления, можно прибегнуть к ключевому слову `const`, когда перечисление определено в файле TypeScript, как показано в листинге 9.24.

ПРИМЕЧАНИЕ

Константные перечисления более строгие, чем обычные перечисления, и все их значения должны быть заданы константными выражениями. Самый простой способ — позволить компилятору присваивать значения или сделать это явно самостоятельно.

Листинг 9.24. Определение перечисления типа const в файле index.ts в папке src

```
const enum Product { Hat, Gloves, Umbrella }
let productValue = Product.Hat;
```

При компиляции кода компилятор встроит каждую ссылку на перечисление, то есть числовое значение будет использоваться напрямую. Если после завершения компиляции открыть файл `index.js` в папке `dist`, то можно увидеть следующий код:

```
...
let productValue = 0 /* Product.Hat */;
...

```

Комментарий добавлен компилятором для указания связи между числовым значением и перечислением. Объект, ранее представлявший перечисление, больше не включается в скомпилированный код.

Константные перечисления могут немного повысить производительность, но это достигается за счет отключения функции поиска имени по значению, как показано в листинге 9.25.

Листинг 9.25. Поиск имени перечисления в файле `index.ts`

```
const enum Product { Hat, Gloves, Umbrella}
let productValue = Product.Hat;
let productName = Product[0];
```

При компиляции кода компилятор выдаст следующую ошибку:

```
src/index.ts(3,27): error TS2476: A const enum member can only be accessed
using a string literal.
```

Объект, используемый для представления обычного перечисления, отвечает за обеспечение возможности поиска и недоступен для константных перечислений.

СОВЕТ

Существует опция компилятора `preserveConstEnums`, которая указывает компилятору генерировать объект даже для константных перечислений. Эта возможность предназначена только для отладки и не восстанавливает функцию поиска.

9.5. ТИПЫ С ЛИТЕРАЛЬНЫМ ЗНАЧЕНИЕМ

Тип с литеральным значением задает определенный набор значений и допускает только эти значения. В результате набор значений рассматривается как отдельный тип, что в целом полезно, но может вызвать путаницу, поскольку размыт границу между типами и значениями. Эту особенность легче всего понять на примере (листинг 9.26).

Листинг 9.26. Использование типа с литеральным значением в файле `index.ts`

```
let restrictedValue: 1 | 2 | 3 = 3;
console.log('Value: ${restrictedValue}');
```

Литеральный тип похож на объединение типов, но вместо типов данных используются литеральные значения, как показано на рис. 9.4.

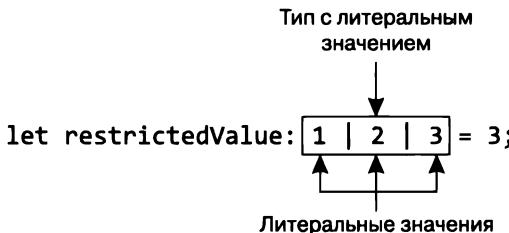


Рис. 9.4. Тип с литеральным значением

Тип с литеральным значением в листинге 9.26 сообщает компилятору, что переменной `restrictedValue` можно присвоить только 1, 2 или 3. Если переменной присваивается любое другое значение, в том числе и числовое, то компилятор выдаст ошибку, как показано в листинге 9.27.

Листинг 9.27. Присвоение другого значения в файле index.ts

```
let restrictedValue: 1 | 2 | 3 = 100;
console.log('Value: ${restrictedValue}');
```

Здесь компилятор определяет, что `100` не входит в список допустимых значений, и выдает следующую ошибку:

```
src/index.ts(1,5): error TS2322: Type '100' is not assignable to type
'1 | 2 | 3'.
```

Комбинация значений рассматривается как отдельный тип, а каждая комбинация литеральных значений — как другой тип, как показано в листинге 9.28. Однако значение одного типа может быть присвоено другому, если оно является одним из разрешенных значений.

Листинг 9.28. Определение второго типа с литеральным значением в файле index.ts

```
let restrictedValue: 1 | 2 | 3 = 1;

let secondValue: 1 | 10 | 100 = 1;

restrictedValue = secondValue;
secondValue = 100;
restrictedValue = secondValue;

console.log('Value: ${restrictedValue}');
```

Первый оператор, присваивающий `secondValue` значению `restrictedValue`, разрешен, поскольку значение `secondValue` является одним из литеральных значений `restrictedValue`. Второй оператор присваивания недопустим, так как значение выходит за пределы допустимого множества, что приводит к следующей ошибке при компиляции кода:

```
src/index.ts(7,1): error TS2322: Type '100' is not assignable to type
'1 | 2 | 3'
```

9.5.1. Типы с литеральным значением в функциях

Типы с литеральным значением наиболее полезны при работе с функциями, поскольку они позволяют ограничивать параметры или результаты определенным набором значений, как показано в листинге 9.29.

Листинг 9.29. Ограничение функции в файле index.ts

```
function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

let total = calculatePrice(2, 19.99);
console.log('Price: ${total}');
```

Параметр quantity функции принимает только значения 1 или 2, и попытка передать любое другое, даже если оно является числом, приведет к ошибке компилятора. Код в листинге 9.29 при компиляции и выполнении выдает следующий результат:

```
Price: 39.98
```

9.5.2. Смешивание типов значений в типе с литеральным значением

Тип с литеральным значением может объединять различные значения, выраженные в литературной форме, включая перечисления (листинг 9.30).

Листинг 9.30. Смешивание значений в типе с литеральным значением в файле index.ts из папки src

```
function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

let total = calculatePrice(2, 19.99);
console.log('Price: ${total}');

function getRandomValue(): 1 | 2 | 3 | 4 {
    return Math.floor(Math.random() * 4) + 1 as 1 | 2 | 3 | 4;
}

enum City { London = "LON", Paris = "PAR", Chicago = "CHI" }

function getMixedValue(): 1 | "Hello" | true | City.London {
    switch (getRandomValue()) {
        case 1:
            return 1;
        case 2:
            return "Hello";
        case 3:
            return true;
```

```

        case 4:
            return City.London;
    }
}

console.log('Value: ${getMixedValue()}');

```

Функция `getRandomValue` возвращает одно из четырех значений, которые используются функцией `getMixedValue` для получения результата. Функция `getMixedValue` показывает, как тип с литеральным значением может объединять значения различных типов, таких как число, строка, логическое значение и значение перечисления. Результат выполнения кода из листинга 9.30 может отличаться, так как значение функции `getMixedValue` выбирается случайным образом:

```

Price: 39.98
Value: true

```

СОВЕТ

Типы с литеральным значением могут использоваться в объединениях типов с обычными типами, создавая комбинации, допускающие конкретные значения одного типа с любыми разрешенными значениями другого. Например, объединению типов `string | true | 3` может быть присвоено любое строковое значение, логическое `true` и числовое `3`.

9.5.3. Переопределения типов с литеральными значениями

В главе 8 мы подробно разобрали, как можно определить связь между типами параметров и возвращаемыми значениями функции с помощью переопределений типов, ограничивающих эффект от использования объединений типов. Переопределения типов можно применять и к типам с литеральным значением (листинг 9.31), которые, по сути, являются объединениями для отдельных значений.

Листинг 9.31. Переопределение типов с литеральным значением в файле index.ts

```

function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

let total = calculatePrice(2, 19.99);
console.log('Price: ${total}');

function getRandomValue(): 1 | 2 | 3 | 4 {
    return Math.floor(Math.random() * 4) + 1 as 1 | 2 | 3 | 4;
}

enum City { London = "LON", Paris = "PAR", Chicago = "CHI" }

function getMixedValue(input: 1);
function getMixedValue(input: 2 | 3): "Hello" | true;
function getMixedValue(input: 4): City.London;

```

```

function getMixedValue(input: number): 1 | "Hello" | true | City.London {
    switch (input) {
        case 1:
            return 1;
        case 2:
            return "Hello";
        case 3:
            return true;
        case 4:
        default:
            return City.London;
    }
}

let first = getMixedValue(1);
let second = getMixedValue(2);
let third = getMixedValue(4);
console.log(${first}, ${second}, ${third});

```

Каждое сопоставление создает связь между параметром и результатом, которые могут быть выражены в виде одного или нескольких значений. Компилятор TypeScript может использовать перегрузки для определения типов переменных `first`, `second` и `third`, что видно по содержимому файла `index.d.ts` в папке `dist`.

```

...
declare let first: 1;
declare let second: true | "Hello";
declare let third: City.London;
...

```

В большинстве проектов подобная возможность не понадобится, но я продемонстрировал ее здесь, чтобы показать, что работа с типами с литеральными значениями аналогична работе с обычными типами. Кроме того, это интересное представление о том, как работает компилятор TypeScript. Код в листинге 9.31 выдает следующий результат:

```

Price: 39.98
1, Hello, LON

```

9.5.4. Шаблонный литеральный строковый тип

Литералы могут использоваться с функцией шаблонной строки JavaScript для создания шаблонов, которые принимают только определенные значения. Это может оказаться лаконичным способом выражения сложных комбинаций значений. В листинге 9.32 создается шаблонная строка, использующая тип с литеральным значением.

Листинг 9.32. Использование типа с литеральным значением в шаблоне в файле `index.ts`

```

function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

```

```

let total = calculatePrice(2, 19.99);
console.log('Price: ${total}');

function getRandomValue(): 1 | 2 | 3 | 4 {
    return Math.floor(Math.random() * 4) + 1 as 1 | 2 | 3 | 4;
}

function getCityString(city: "London" | "Paris" | "Chicago")
    : 'City: ${"London" | "Paris" | "Chicago"}' {
    return 'City: ${city}';
}

let str = getCityString("London");
console.log(str);

```

Функция `getCityString` определяет параметр, ограниченный тремя строковыми значениями типа с литеральным значением. Результат работы функции выражается с помощью шаблонной строки, использующей тип с литеральным значением, следующим образом:

```

...
'City: ${"London" | "Paris" | "Chicago"}'
...

```

Взгляните на содержимое файла `index.d.ts` в папке `dist`. В нем вы увидите, как компилятор TypeScript определяет тип для переменной `str`:

```

...
declare let str: "City: London" | "City: Paris" | "City: Chicago";
...

```

Компилятор использовал тип с литеральным значением для расширения шаблонной строки до полного набора строк, которые могут быть присвоены переменной `str`. Код в листинге 9.32 выдает следующий результат:

```

Price: 39.98
City: London

```

9.6. ПСЕВДОНИМЫ ТИПОВ

Чтобы избежать повторений, в TypeScript предусмотрена функция псевдонимов типов. Она позволяет присвоить пользовательской комбинации типов имя и применять его там, где это необходимо, как показано в листинге 9.33.

Листинг 9.33. Использование псевдонимов типов в файле `index.ts`

```

function calculatePrice(quantity: 1 | 2, price: number): number {
    return quantity * price;
}

let total = calculatePrice(2, 19.99);
console.log('Price: ${total}');

```

```

type numVals = 1 | 2 | 3 | 4;

function getRandomValue(): numVals {
    return Math.floor(Math.random() * 4) + 1 as numVals;
}

type cities = "London" | "Paris" | "Chicago";
type cityResponse = 'City: ${ cities }';

function getCityString(city: cities): cityResponse {
    return 'City: ${city}';
}

let str = getCityString("London");
console.log(str);

```

Псевдонимы типов очищают код TypeScript, сокращая дублирование. Например, вместо повторного описания набора `cities` для параметра и результата функции `getCityString` удобнее создать псевдоним типа, который можно использовать как для параметра функции, так и в строке шаблона:

```

...
type cities = "London" | "Paris" | "Chicago";
type cityResponse = 'City: ${ cities }';
...

```

Псевдонимы типов определяются с помощью ключевого слова `type`, за которым следует имя псевдонима, знак равенства и тип, который будет псевдонимом, как показано на рис. 9.5.

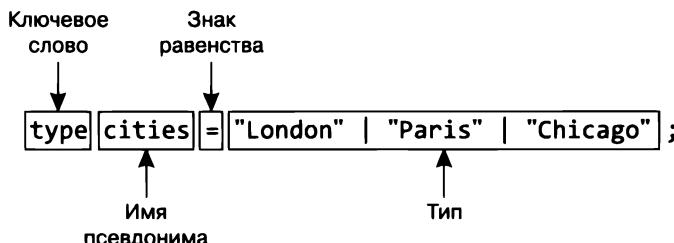


Рис. 9.5. Определение псевдонима типа

Имя, присвоенное псевдониму, используется вместо полного описания типа. Псевдонимы типа позволяют упростить ссылки на сложный тип или комбинации типов, не влияя при этом на работу компилятора TypeScript с типом. Псевдоним может использоваться в аннотациях или утверждениях типа как обычно. Код в листинге 9.33 при компиляции и выполнении выдает следующий результат:

```

Price: 39.98
City: London

```

РЕЗЮМЕ

В этой главе вы узнали, как в TypeScript работать с массивами, кортежами и перечислениями, которые реализуются компилятором TypeScript. Также познакомились с тем, как определять типы с литеральным значением и как использовать псевдонимы для последовательного описания типов.

- К массивам можно применять аннотации для ограничения типов и их содержимого.
- Компилятор TypeScript может самостоятельно определить тип массива по его начальному содержимому.
- Компилятор TypeScript позволяет пустым массивам, определенным без аннотации типа, принимать любое значение.
- Базовые кортежи представляют собой массивы фиксированной длины, где каждый элемент имеет свою аннотацию типа, однако кортежи могут также содержать необязательные и rest-элементы.
- Перечисления позволяют представить коллекцию значений последовательно. Они полностью реализуются компилятором TypeScript, что может привести к некоторым странностям в генерируемом JavaScript-коде.
- Типы с литеральным значением представляют собой определенный набор значений, причем только те, которые соответствуют данному типу.
- Псевдонимы типов позволяют присвоить имя типу, например объединению, чтобы обеспечить согласованность и избежать дублирования.

В следующей главе мы познакомимся с возможностями, которые предоставляет TypeScript для работы с объектами.

10

Объекты

В этой главе

- ✓ Использование структур типов для описания объектов.
- ✓ Упрощение использования структур типов с помощью псевдонимов.
- ✓ Создание объединений структур типов.
- ✓ Защита структур типов.
- ✓ Создание и применение пересечений типов.

В данной главе описывается работа TypeScript с объектами. Как объяснялось в главах 3 и 4, JavaScript отличается гибким подходом к работе с объектами, а TypeScript стремится найти баланс между предотвращением наиболее распространенных ошибок и сохранением полезных возможностей. Эта тема продолжится в главе 11, где рассказывается о поддержке TypeScript для классов. В табл. 10.1 приведено краткое содержание главы.

Таблица 10.1. Краткое содержание главы

Задача	Решение	Листинг
Описать объект компилятору TypeScript	Используйте структуру типа	4–6, 8
Описать типы неправильных структур	Используйте необязательные свойства	7, 9, 10

Продолжение ↓

Таблица 10.1 (продолжение)

Задача	Решение	Листинг
Использовать ту же структуру для описания нескольких объектов	Воспользуйтесь псевдонимом типа	11
Объединить типы структур	Используйте объединение или пересечение типов	12, 13, 17–23
Применить защиту типа для типов объектов	Проверьте свойства, определяемые объектом, с помощью ключевого слова <code>in</code>	14, 15
Повторно использовать защиту типа	Определите предикатную функцию	16

В качестве краткой справки в табл. 10.2 перечислены параметры компилятора TypeScript, используемые в данной главе.

Таблица 10.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
<code>target</code>	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода
<code>outDir</code>	Задает каталог, в который будут помещены файлы JavaScript
<code>rootDir</code>	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
<code>declaration</code>	При включении этого параметра компилятор создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода. Более подробно эти файлы описаны в главе 15
<code>strictNullChecks</code>	Не допускает присваивания <code>null</code> и <code>undefined</code> в качестве значений для других типов

10.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В текущей главе мы продолжим использовать проект `types`, созданный в главе 7 и обновленный в последующих главах. Чтобы приступить к работе с материалом из этой главы, замените содержимое файла `index.ts` в папке `src` кодом, показанным в листинге 10.1.

Листинг 10.1. Замена содержимого файла `index.ts` из папки `src`

```
let hat = { name: "Hat", price: 100 };
let gloves = { name: "Gloves", price: 75 };

let products = [hat, gloves];

products.forEach(prod => console.log(`${prod.name}: ${prod.price}`));
```

Сбросьте конфигурацию компилятора, заменив содержимое файла `tsconfig.json` настройками, показанными в листинге 10.2.

Листинг 10.2. Настройка компилятора в файле `tsconfig.json` из папки `types`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    // "strictNullChecks": true,
  }
}
```

В конфигурацию компилятора входит опция `declaration`, которая означает, что компилятор будет создавать файлы декларации типов вместе с файлами JavaScript. Настоящее назначение файлов декларации описано в главе 15, но в этой главе они будут использоваться для объяснения того, как компилятор работает с типами данных.

С помощью консоли выполните в папке `types` команду из листинга 10.3 для запуска компилятора TypeScript, чтобы он автоматически выполнял код после его компиляции.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 10.3. Запуск компилятора TypeScript

```
npm start
```

Компилятор скомпилирует проект, выполнит вывод, а затем перейдет в режим наблюдения, выдав следующий результат:

```
7:10:34 AM - Starting compilation in watch mode...
7:10:35 AM - Found 0 errors. Watching for file changes.
Hat: 100
Gloves: 75
```

10.2. РАБОТА С ОБЪЕКТАМИ

Объекты JavaScript – это коллекции свойств, которые могут быть созданы с помощью литерального синтаксиса, функций-конструкторов или классов. Независимо от способа создания, объекты можно изменять, добавляя или удаляя свойства, и получать значения различных типов. Для обеспечения типизации объектов в TypeScript основное внимание уделяется их «структуре», представляющей собой комбинацию имен свойств и их типов.

Компилятор TypeScript стремится к единообразному использованию объектов, определяя их общие характеристики. Лучший способ понять, как это работает, — посмотреть на файлы деклараций, которые компилятор создает при включенной опции `declarations`. Например, в файле `index.d.ts` в папке `dist` компилятор использует структуру каждого объекта, определенного в листинге 10.1, в качестве его типа, например, так:

```
declare let hat: { name: string; price: number; };
declare let gloves: { name: string; price: number; };
declare let products: { name: string; price: number; }[];
```

Я отформатировал содержимое файла декларации, чтобы было легче увидеть, как компилятор определяет тип каждого объекта по его структуре. Структура объектов, помещенных в массив, используется для определения соответствующего типа всего массива, и этот подход позволяет избежать распространенных ошибок. В листинге 10.4 добавляется объект другой структуры.

Листинг 10.4. Добавление объекта в файл `index.ts` из папки `src`

```
let hat = { name: "Hat", price: 100 };
let gloves = { name: "Gloves", price: 75 };
let umbrella = { name: "Umbrella" };

let products = [hat, gloves, umbrella];

products.forEach(prod => console.log(`"${prod.name}": ${prod.price}`));
```

Несмотря на то что объекты в листинге 10.4 определены с использованием лiteralного синтаксиса, компилятор TypeScript может обнаружить несоответствия при использовании объектов. Например, объект `umbrella` не содержит свойства `price`, и компилятор при компиляции выдаст следующую ошибку:

```
src/index.ts(9,60): error TS2339: Property 'price' does not exist on type
'{ name: string; }'.
```

Это происходит потому, что стрелочная функция, используемая в методе `forEach`, пытается считать свойство `price`, которое отсутствует у некоторых объектов в массиве `products`. Компилятор правильно определяет структуру объектов в данном примере, как видно из файла `index.d.ts` в папке `dist`.

```
declare let hat: { name: string; price: number; };
declare let gloves: { name: string; price: number; };
declare let umbrella: { name: string; };
declare let products: { name: string; }[];
```

Обратите внимание, что тип массива `products` изменился. Когда объекты разной структуры используются вместе, скажем, в массиве, компилятор создает тип, который имеет общие свойства содержащихся в нем объектов, поскольку именно с ними можно безопасно работать. В нашем случае единственным общим свойством для всех объектов массива является строковое свойство `name`, поэтому компилятор выдает ошибку при попытке доступа к свойству `price`.

10.2.1. Использование аннотаций структуры типа объекта

Для объектных литералов компилятор TypeScript определяет тип каждого свойства по присвоенному ему значению. Типы также могут быть явно указаны с помощью аннотаций типов, которые применяются к отдельным свойствам, как показано в листинге 10.5.

Листинг 10.5. Использование аннотаций структуры типа объекта в файле index.ts

```
let hat = { name: "Hat", price: 100 };
let gloves = { name: "Gloves", price: 75 };
let umbrella = { name: "Umbrella" };

let products: { name: string, price: number }[] = [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price}`));
```

Аннотация типа ограничивает содержимое массива `products` объектами, имеющими типы свойств `name` и `price`, которые являются строковыми и числовыми значениями, как показано на рис. 10.1.

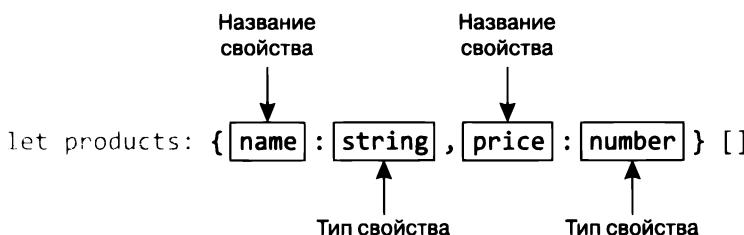


Рис. 10.1. Структура типа объекта

Компилятор по-прежнему сообщает об ошибке для кода в листинге 10.5, но теперь проблема заключается в том, что объект `umbrella` не соответствует структуре, указанной в аннотации типа для массива `products`, что дает более полезное описание проблемы.

```
src/index.ts(5,64): error TS2741: Property 'price' is missing in type
'{ name: string; }' but required in type '{ name: string; price:
number; }'.
```

10.2.2. О соответствии структур типов

Чтобы соответствовать типу, объект должен определять все свойства структуры. Компилятор все равно сопоставит объект, если у него есть дополнительные свойства, не определенные структурой типа, как показано в листинге 10.6.

Листинг 10.6. Добавление свойств в файл index.ts в папке src

```
let hat = { name: "Hat", price: 100 };
let gloves = { name: "Gloves", price: 75 };
let umbrella = { name: "Umbrella", price: 30, waterproof: true };

let products: { name: string, price?: number }[] = [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price}`));
```

Новые свойства позволяют объекту `umbrella` соответствовать структуре типа массива, поскольку теперь он определяет свойства `name` и `price`. Свойство `waterproof` игнорируется, так как оно не является частью структуры типа. При компиляции и выполнении кода в листинге 10.6 получается следующий код:

```
Hat: 100
Gloves: 75
Umbrella: 30
```

Обратите внимание, что для указания на то, что отдельные объекты имеют определенную структуру, аннотации типов не требуются. Компилятор TypeScript автоматически определяет соответствие объекта структуре, анализируя его свойства и их значения.

Дополнительные свойства для неправильных структур

Необязательные свойства делают структуру типа более гибкой, позволяя ей соответствовать объектам, не обладающим этими свойствами (листинг 10.7). Это может быть важно при работе с набором объектов разной структуры, и при этом необходимо использовать свойство, если оно имеется.

Листинг 10.7. Использование необязательного свойства в файле index.ts из папки src

```
let hat = { name: "Hat", price: 100 };
let gloves = { name: "Gloves", price: 75 };
let umbrella = { name: "Umbrella", price: 30, waterproof: true };

let products: { name: string, price?: number, waterproof?: boolean }[]
= [hat, gloves, umbrella];

products.forEach(prod =>
  console.log(`${prod.name}: ${prod.price} '
+ 'Waterproof: ${ prod.waterproof }`));
```

Необязательные свойства задаются с помощью того же синтаксиса, как и необязательные параметры функций, где после имени свойства ставится вопросительный знак (рис. 10.2).

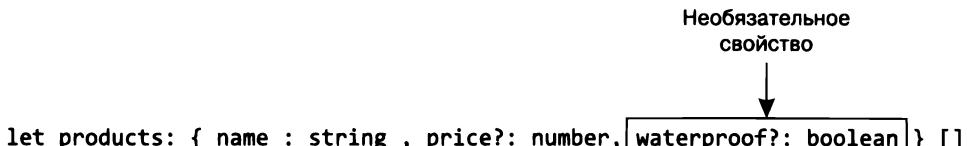


Рис. 10.2. Необязательное свойство в структуре типа

Структура типа с необязательными свойствами может соответствовать объектам, которые не определяют эти свойства, при условии, что обязательные свойства определены. При использовании необязательного свойства, как, например, в функции `forEach` в листинге 10.7, значение необязательного свойства будет либо значением, определенным объектом, либо `undefined`, как показано в следующем выводе кода:

```
Hat: 100 Waterproof: undefined
Gloves: 75 Waterproof: undefined
Umbrella: 30 Waterproof: true
```

Объекты `hat` и `gloves` не определяют необязательное свойство `waterproof`, поэтому значение, получаемое в функции `forEach`, — `undefined`. Объект `umbrella` определяет это свойство, и его значение отображается.

Включение методов в структуры типа

Структуры типа могут включать в себя как методы, так и свойства, что предоставляет больше возможностей для контроля над сопоставлением объектов с типом, как показано в листинге 10.8.

Листинг 10.8. Включение метода в структуру типа в файле index.ts

```
enum Feature { Waterproof, Insulated }

let hat = { name: "Hat", price: 100 };
let gloves = { name: "Gloves", price: 75 };
let umbrella = { name: "Umbrella", price: 30,
    hasFeature: (feature) => feature === Feature.Waterproof };

let products: { name: string, price?: number,
    hasFeature?(Feature): boolean }[]
= [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price} '
+ 'Waterproof: ${prod.hasFeature(Feature.Waterproof)}'));
```

Аннотация типа для массива `products` включает необязательное свойство `hasFeature`, которое представляет собой метод. Формат свойства метода подобен обычному свойству, но с добавлением круглых скобок, описывающих типы параметров, после которых ставится двоеточие, а затем указывается тип возвращаемого значения, как показано на рис. 10.3.

```
let products: { name : string , price?: number,
    hasFeature ? (Feature) : boolean } [ ]
```

↑
Имя метода ↑
Тип параметра ↑
Тип результата

Рис. 10.3. Метод в структуре типа

Метод, включенный в структуру типа в листинге 10.8, определяет метод `hasFeature`, имеющий один параметр, который должен быть значением из

перечисления `Feature` (также определенного в листинге 10.8), и возвращает результат типа `boolean`.

СОВЕТ

Методы в структурах типа не обязательно должны быть опциональными. Однако если они таковыми являются, как в листинге 10.8, то знак вопроса ставится после имени метода и перед круглыми скобками, обозначающими начало типов параметров.

Объект `umbrella` определяет метод `hasFeature` с правильными типами, но поскольку этот метод является необязательным, то объекты `hat` и `gloves` также соответствуют этой структуре типа. Необязательные методы определяются как `undefined`, если они не присутствуют в объекте, поэтому при компиляции и выполнении кода в листинге 10.8 возникает следующая ошибка:

```
C:\types\dist\index.js:12
  + 'Waterproof: ${prod.hasFeature(Feature.Waterproof)}'))';
TypeError: prod.hasFeature is not a function
```

Как и в случае с обычными свойствами, перед вызовом метода необходимо убедиться, что он реализован.

Обеспечение строгой проверки методов

Для предотвращения ошибок, подобных той, что была описана в предыдущем разделе, компилятор TypeScript может сообщать о них, когда необязательный метод, заданный структурой типа, используется без предварительной проверки на наличие значений типа `undefined`. Эта проверка включается параметром `strictNullChecks`, который также использовался в предыдущих главах. Измените конфигурацию компилятора, включив настройки, как показано в листинге 10.9.

Листинг 10.9. Настройка компилятора в файле `tsconfig.json` из папки `types`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true,
  }
}
```

После сохранения файла конфигурации компилятор пересоберет проект и выдаст следующую ошибку:

```
src/index.ts(13,22): error TS2722: Cannot invoke an object which is
possibly 'undefined'.
```

Данная ошибка предотвращает использование необязательных методов до тех пор, пока они не будут проверены на предмет их существования в объекте, как показано в листинге 10.10.

Листинг 10.10. Проверка наличия необязательного метода в файле index.ts из папки src

```
enum Feature { Waterproof, Insulated }

let hat = { name: "Hat", price: 100 };
let gloves = { name: "Gloves", price: 75 };
let umbrella = { name: "Umbrella", price: 30,
    hasFeature: (feature) => feature === Feature.Waterproof };

let products: { name: string, price?: number,
    hasFeature?(Feature): boolean }[]
= [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price} ' +
    '${ prod.hasFeature ? prod.hasFeature(Feature.Waterproof) : "false" }'
));
```

Метод `hasFeature` вызывается только в том случае, если он был определен. Код в листинге 10.10 при компиляции и выполнении выдает следующий результат:

```
Hat: 100 false
Gloves: 75 false
Umbrella: 30 true
```

10.2.3. Использование псевдонимов типов для структур типов

Псевдоним типа позволяет присвоить имя конкретной структуре, что упрощает регулярное обращение к ней в коде (листинг 10.11). Присвоенное структуре имя может быть использовано в аннотациях типов.

Листинг 10.11. Использование псевдонима для структуры типа в файле index.ts

```
enum Feature { Waterproof, Insulated }

type Product = {
    name: string,
    price?: number,
    hasFeature?(Feature): boolean
};

let hat = { name: "Hat", price: 100 };
let gloves = { name: "Gloves", price: 75 };
let umbrella = { name: "Umbrella", price: 30,
    hasFeature: (feature) => feature === Feature.Waterproof };

let products: Product[] = [hat, gloves, umbrella];

products.forEach(prod => console.log(`${prod.name}: ${prod.price} ' +
    '${ prod.hasFeature ? prod.hasFeature(Feature.Waterproof) : "false" }'
));
```

Псевдоним присваивает форме имя, которое можно использовать в аннотациях типов. Здесь создан псевдоним `Product`, который используется в качестве типа для массива. Псевдоним не изменяет вывод кода при его компиляции и выполнении.

```
Hat: 100 false
Gloves: 75 false
Umbrella: 30 true
```

10.2.4. Объединения структур типов

В главе 7 мы рассмотрели функцию, которая позволяет объединять различные типы вместе, давая возможность массивам или параметрам функций принимать несколько типов. Как уже было объяснено, объединенные типы являются самостоятельными типами и содержат свойства, определяемые всеми входящими в них типами. Это не очень полезно при работе с объединениями примитивных типов данных, поскольку общих свойств мало. Однако это становится более полезным при взаимодействии с объектами, как показано в листинге 10.12.

Листинг 10.12. Использование объединения типов в файле `index.ts`

```
type Product = {
    id: number,
    name: string,
    price?: number
};

type Person = {
    id: string,
    name: string,
    city: string
};

let hat = { id: 1, name: "Hat", price: 100 };
let gloves = { id: 2, name: "Gloves", price: 75 };
let umbrella = { id: 3, name: "Umbrella", price: 30 };
let bob = { id: "bsmith", name: "Bob", city: "London" };

let dataItems: (Product | Person)[] = [hat, gloves, umbrella, bob];

dataItems.forEach(item =>
    console.log('ID: ${item.id}, Name: ${item.name}'));
```

В данном примере массив `dataItems` был аннотирован объединением типов `Product` и `Person`. Эти типы имеют два общих свойства — `id` и `name`, которые можно использовать при обработке массива без необходимости сужения до одного типа.

```
...
dataItems.forEach(item =>
    console.log('ID: ${item.id}, Name: ${item.name}'));
...
```

Это единственныe свойства, к которым можно получить доступ, так как они едины для всех типов в объединении. Любая попытка получить доступ к свойству

price, определяемому типом `Product`, или к свойству `city`, определяемому типом `Person`, приведет к ошибке, поскольку эти свойства не входят в объединение `Product | Person`. Код в листинге 10.12 выдает следующий результат:

```
ID: 1, Name: Hat
ID: 2, Name: Gloves
ID: 3, Name: Umbrella
ID: bsmith, Name: Bob
```

10.2.5. Объединение типов свойств

Когда создается объединение структур типов, объединяются также и общие свойства. Данный эффект легче понять, создав тип, эквивалентный объединению, как показано в листинге 10.13.

Листинг 10.13. Создание эквивалентного типа в файле `index.ts`

```
type Product = {
    id: number,
    name: string,
    price?: number
};

type Person = {
    id: string,
    name: string,
    city: string
};

type UnionType = {
    id: number | string,
    name: string
};

let hat = { id: 1, name: "Hat", price: 100 };
let gloves = { id: 2, name: "Gloves", price: 75 };
let umbrella = { id: 3, name: "Umbrella", price: 30 };
let bob = { id: "bsmith", name: "Bob", city: "London" };

let dataItems: UnionType[] = [hat, gloves, umbrella, bob];

dataItems.forEach(item =>
    console.log('ID: ${item.id}, Name: ${item.name}'));
```

В нашем примере `UnionType` демонстрирует эффект объединения типов `Product` и `Person`. Тип свойства `id` представляет собой объединение `number | string`, так как свойство `id` в типе `Product` является числом, а в типе `Person` — строкой. Свойство `name` в обоих типах — строка, поэтому в объединении используется именно этот тип для свойства `name`. Код в листинге 10.13 при компиляции и выполнении выдает следующий результат:

```
ID: 1, Name: Hat
ID: 2, Name: Gloves
ID: 3, Name: Umbrella
ID: bsmith, Name: Bob
```

10.2.6. Защита типов для объектов

В предыдущем разделе было показано, как объединения структур типов могут быть полезны сами по себе, однако для доступа к определенному типу и ко всем определяемым им функциям все равно необходимо использовать защиту типов.

В главе 7 был продемонстрирован пример использования ключевого слова `typeof` для создания защиты типа. Ключевое слово `typeof` — это стандартная функция JavaScript, которую компилятор TypeScript распознает и применяет в процессе проверки типов. Однако `typeof` нельзя использовать с объектами, поскольку оно всегда возвращает один и тот же результат, как показано в листинге 10.14.

Листинг 10.14. Защита типов в файле `index.ts` из папки `src`

```
type Product = {
    id: number,
    name: string,
    price?: number
};

type Person = {
    id: string,
    name: string,
    city: string
};

let hat = { id: 1, name: "Hat", price: 100 };
let gloves = { id: 2, name: "Gloves", price: 75 };
let umbrella = { id: 3, name: "Umbrella", price: 30 };
let bob = { id: "bsmith", name: "Bob", city: "London" };

let dataItems: (Product | Person)[] = [hat, gloves, umbrella, bob];

dataItems.forEach(item =>
    console.log('ID: ${item.id}, Type: ${typeof item}'));

```

В этом листинге тип массива задается как объединение типов `Product` и `Person`, а для определения типа каждого элемента массива используется ключевое слово `typeof` в функции `forEach`, в результате чего получается следующее:

```
ID: 1, Type: object
ID: 2, Type: object
ID: 3, Type: object
ID: bsmith, Type: object
```

Как видно, функция структуры типа полностью обеспечивается TypeScript, и все объекты имеют тип `object`, поскольку JavaScript предоставляет ограниченные возможности в этом контексте. В результате ключевое слово `typeof` бесполезно для определения соответствия объекта структурам `Product` и `Person`.

Защита типов с помощью проверки свойств

Для более точного определения структур типов средствами JavaScript используется простой и надежный метод — добавить ключевое слово `in` для проверки наличия свойства, как показано в листинге 10.15.

Листинг 10.15. Защита типов в файле index.ts

```

type Product = {
    id: number,
    name: string,
    price?: number
};

type Person = {
    id: string,
    name: string,
    city: string
};

let hat = { id: 1, name: "Hat", price: 100 };
let gloves = { id: 2, name: "Gloves", price: 75 };
let umbrella = { id: 3, name: "Umbrella", price: 30 };
let bob = { id: "bsmith", name: "Bob", city: "London" };

let dataItems: (Product | Person)[] = [hat, gloves, umbrella, bob];

dataItems.forEach(item => {
    if ("city" in item) {
        console.log(`Person: ${item.name}: ${item.city}`);
    } else {
        console.log(`Product: ${item.name}: ${item.price}`);
    }
});

```

Задача здесь — определить, что каждый объект в массиве соответствует структуре `Product` или `Person`. Аннотация типа `(Product | Person)[]` подсказывает нам, что только эти два типа могут содержаться в массиве.

Структура объекта представляет собой комбинацию его свойств. Защита типа выполняется путем проверки наличия определенных свойств, которые входят в одну структуру, но не входят в другую. В случае листинга 10.15 любой объект, имеющий свойство `city`, должен соответствовать структуре `Person`, поскольку это свойство не входит в структуру `Product`. Чтобы создать защиту типа, проверяющую наличие свойства, имя свойства выражается в виде строкового литерала. Затем следует ключевое слово `in`, после которого указывается проверяемый объект, как показано на рис. 10.4.

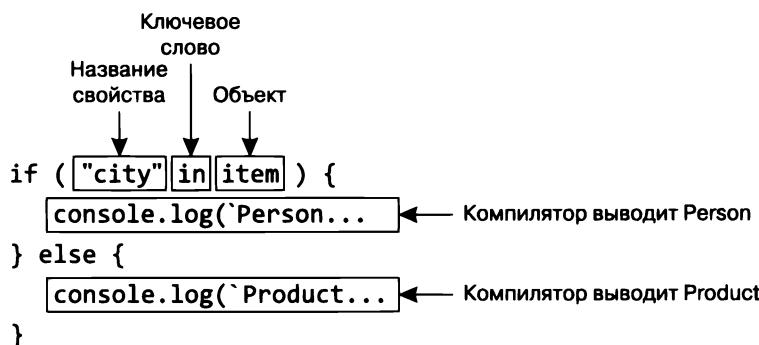


Рис. 10.4. Использование ключевого слова `in`

Выражение `in` возвращает `true` для объектов, определяющих указанное свойство, и `false` в противном случае. Компилятор TypeScript понимает важность проверки на наличие свойства и выводит тип внутри блоков кода оператора `if/else`. Код в листинге 10.15 при компиляции и выполнении выдает следующий результат:

```
Product: Hat: 100
Product: Gloves: 75
Product: Umbrella: 30
Person: Bob: London
```

ПРЕДОТВРАЩЕНИЕ РАСПРОСТРАНЕННЫХ ПРОБЛЕМ С ЗАЩИТОЙ ТИПА

Важно создавать тесты защиты типов, которые однозначно и точно различают типы. Если компилятор выдает неожиданные ошибки при использовании защиты типов, то, скорее всего, причина кроется в неточном teste. Существует две распространенные проблемы, которых следует избегать. Первая — это создание неточного теста, который не позволяет надежно различать типы, как, например, этот тест:

```
dataItems.forEach(item => {
  if ("id" in item && "name" in item) {
    console.log('Person: ${item.name}: ${item.city}');
  } else {
    console.log('Product: ${item.name}: ${item.price}');
  }
});
```

Данный тест проверяет наличие свойств `id` и `name`, но они определяются типами `Person` и `Product`, и тест не предоставляет компилятору достаточно информации для определения конкретного типа. Тип, выводимый в блоке `if`, — это объединение `Product | Person`, а это означает, что использование свойства `city` приведет к ошибке. Тип, выводимый в блоке `else`, — `never`, поскольку все возможные типы уже определены и компилятор выдаст ошибки при использовании свойств `name` и `price`. Аналогичная проблема возникает при тестировании необязательных свойств, например, так:

```
dataItems.forEach(item => {
  if ("price" in item) {
    console.log('Product: ${item.name}: ${item.price}');
  } else {
    console.log('Person: ${item.name}: ${item.city}');
  }
});
```

Тест будет соответствовать объектам, определяющим свойство `price`, что приведет к выводу типа `Product` в блоке `if`, как и предполагалось (обратите внимание, что утверждения в блоках кода в этом примере поменялись местами). Проблема заключается в том, что объекты могут соответствовать структуре `Product`, если у них нет свойства `price`, а значит, тип, определяемый в блоке `else`, будет `Product | Person` и компилятор выдаст ошибку в отношении свойства `city`.

Написание эффективных тестов для типов может потребовать тщательного анализа и тестирования, однако с опытом данный процесс станет проще.

Защита типа с помощью функции предиката типа

Ключевое слово `in` — полезный способ определения, соответствует ли объект конкретной структуре, но он требует написания одних и тех же проверок каждый раз, когда необходимо определить тип. TypeScript также поддерживает защиту типов объектов с помощью функции, как показано в листинге 10.16.

Листинг 10.16. Защита типов с помощью функции в файле index.ts

```
type Product = {
    id: number,
    name: string,
    price?: number
};

type Person = {
    id: string,
    name: string,
    city: string
};

let hat = { id: 1, name: "Hat", price: 100 };
let gloves = { id: 2, name: "Gloves", price: 75 };
let umbrella = { id: 3, name: "Umbrella", price: 30 };
let bob = { id: "bsmith", name: "Bob", city: "London" };
let dataItems: (Product | Person)[] = [hat, gloves, umbrella, bob];

function isPerson(testObj: any): testObj is Person {
    return testObj.city !== undefined;
}

dataItems.forEach(item => {
    if (isPerson(item)) {
        console.log(`Person: ${item.name}: ${item.city}`);
    } else {
        console.log(`Product: ${item.name}: ${item.price}`);
    }
});
```

Защита типов объектов осуществляется с помощью функции, использующей ключевое слово `is`, как показано на рис. 10.5.



Рис. 10.5. Функция защиты объектного типа

Результат функции, представляющий собой *предикат типа*, сообщает компилятору, какой из параметров функции проверяется, и тип, на который проверяется функция. В листинге 10.16 функция `isPerson` проверяет свой параметр `testObj` на тип `Person`. Если результат функции `true`, то компилятор TypeScript отнесет объект к указанному типу.

Использование функции для защиты типов может быть более гибким, поскольку тип параметра может быть `any`, что позволяет проверять свойства без использования строковых литералов и ключевого слова `in`.

СОВЕТ

Нет никаких ограничений на имя функции защиты типа, но принято добавлять префикс `is` к защищаемому типу, например, функция, проверяющая тип `Person`, называется `isPerson`, а проверяющая тип `Product` — `isProduct`.

Код в листинге 10.16 при компиляции и выполнении выдает следующий результат, показывая, что использование функции защиты типов имеет тот же эффект, что и ключевое слово `in`:

```
Product: Hat: 100
Product: Gloves: 75
Product: Umbrella: 30
Person: Bob: London
```

10.3. ПЕРЕСЕЧЕНИЯ ТИПОВ

Пересечения типов позволяют объединить характеристики нескольких типов, давая возможность задействовать все их свойства и методы. В отличие от объединений типов, которые позволяют использовать только общие методы, пересечения допускают совмещение всех функций каждого типа. Пример использования пересечения типов приведен в листинге 10.17.

Листинг 10.17. Определение пересечения типов в файле `index.ts` из папки `src`

```
type Person = {
    id: string,
    name: string,
    city: string
};

type Employee = {
    company: string,
    dept: string
};

let bob = { id: "bsmith", name: "Bob", city: "London",
    company: "Acme Co", dept: "Sales" };

let dataItems: (Person & Employee)[] = [bob];
```

```
dataItems.forEach(item => {
  console.log('Person: ${item.id}, ${item.name}, ${item.city}');
  console.log('Employee: ${item.id}, ${item.company}, ${item.dept}');
});
```

Тип массива `dataItems` устанавливается как пересечение типов `Person` и `Employee`. Пересечения определяются с помощью амперсанда (`&`) между двумя или более типами, как показано на рис. 10.6.

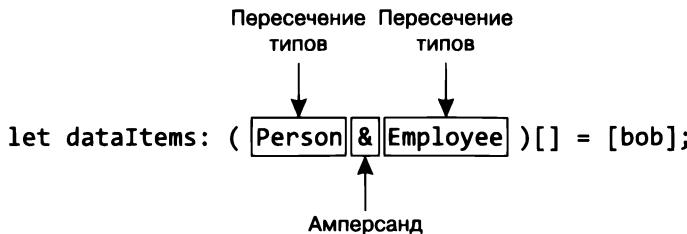


Рис. 10.6. Определение пересечения типов

Объект будет соответствовать структуре пересечения типов только в том случае, если он определяет все свойства, входящие в объединение всех типов в этом пересечении, как показано на рис. 10.7.

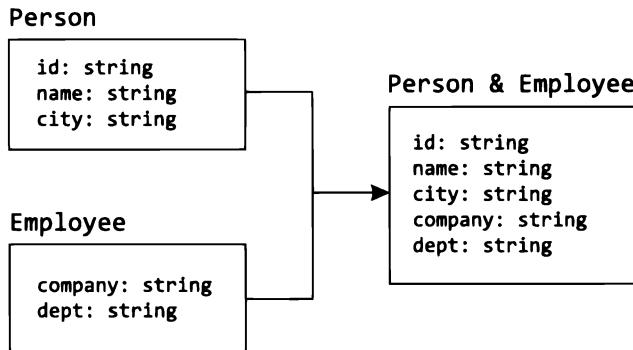


Рис. 10.7. Эффект пересечения типов

В листинге 10.17 пересечение типов `Person` и `Employee` приводит к тому, что массив `dataItems` может содержать только объекты, определяющие свойства `id`, `name`, `city`, `company` и `dept`.

Содержимое массива обрабатывается с помощью метода `forEach`, что демонстрирует возможность использования свойств обоих типов в пересечении. Код в листинге при компиляции и выполнении выдает следующий результат:

```
Person: bsmith, Bob, London
Employee: bsmith, Acme Co, Sales
```

10.3.1. Использование пересечений для сопоставления данных

Пересечения полезны в тех случаях, когда требуется извлечь данные из одного источника и внедрить новую функциональность для их использования в других частях приложения, или при необходимости сопоставления и объединения объектов из двух источников данных. В JavaScript существует простой способ интегрировать функциональность одного объекта в другой, а пересечения типов позволяют явно описать используемые типы, чтобы компилятор TypeScript смог их проверить. В листинге 10.18 показана функция, которая сопоставляет два массива данных.

Листинг 10.18. Корреляция данных в файле index.ts из папки src

```
type Person = {
    id: string,
    name: string,
    city: string
};

type Employee = {
    id: string,
    company: string,
    dept: string
};

type EmployedPerson = Person & Employee;

function correlateData(peopleData: Person[], staff: Employee[])
    : EmployedPerson[] {
    const defaults = { company: "None", dept: "None" };
    return peopleData.map(p => ({ ...p,
        ...staff.find(e => e.id === p.id) || { ...defaults, id: p.id } }));
}

let people: Person[] =
    [{ id: "bsmith", name: "Bob Smith", city: "London" },
     { id: "ajones", name: "Alice Jones", city: "Paris" },
     { id: "dpeters", name: "Dora Peters", city: "New York" }];

let employees: Employee[] =
    [{ id: "bsmith", company: "Acme Co", dept: "Sales" },
     { id: "dpeters", company: "Acme Co", dept: "Development" }];

let dataItems: EmployedPerson[] = correlateData(people, employees);

dataItems.forEach(item => {
    console.log('Person: ${item.id}, ${item.name}, ${item.city}');
    console.log('Employee: ${item.id}, ${item.company}, ${item.dept}');
});
```

В данном примере функция `correlateData` получает массив объектов `Person` и массив объектов `Employee`. Она использует общее свойство `id` для создания объектов, объединяющих свойства обеих структур типов. Каждый объект `Person` обрабатывается методом `map`, при этом метод `find` массива используется для поиска

объекта `Employee` с тем же значением `id`. Оператор расширения объекта (`spread`) используется для создания нового объекта, соответствующего структуре пересечения. Поскольку в результатах функции `correlateData` должны быть определены все свойства пересечения, используются значения по умолчанию, когда нет подходящего объекта `Employee`.

```
...
const defaults = { company: "None", dept: "None"};
return peopleData.map(p => ({ ...p,
    ...staff.find(e => e.id === p.id) || { ...defaults, id: p.id } }));
...
...
```

В листинге 10.18 применялись аннотации типов, чтобы облегчить понимание цели кода, но код будет работать и без них. Компилятор TypeScript анализирует эффекты утверждений кода и может понять, что эффект этого утверждения заключается в создании объектов, соответствующих структуре пересечения типов.

Код в листинге 10.18 при компиляции и выполнении выдает следующий результат:

```
Person: bsmith, Bob Smith, London
Employee: bsmith, Acme Co, Sales
Person: ajones, Alice Jones, Paris
Employee: ajones, None, None
Person: dpeters, Dora Peters, New York
Employee: dpeters, Acme Co, Development
```

10.3.2. Объединение пересечений

Поскольку пересечение объединяет признаки нескольких типов, объект, соответствующий структуре пересечения, также удовлетворяет каждому из типов, входящих в пересечение. Например, объект, соответствующий типу `Person & Employee`, может использоваться там, где указан тип `Person` или тип `Employee`, как показано в листинге 10.19.

Листинг 10.19. Использование типов в пересечении в файле `index.ts`

```
type Person = {
    id: string,
    name: string,
    city: string
};

type Employee = {
    id: string,
    company: string,
    dept: string
};

type EmployedPerson = Person & Employee;

function correlateData(peopleData: Person[], staff: Employee[])
    : EmployedPerson[] {
    const defaults = { company: "None", dept: "None"};
```

```

        return peopleData.map(p => ({ ...p,
            ...staff.find(e => e.id === p.id) || { ...defaults, id: p.id } }));
    }

let people: Person[] =
    [{ id: "bsmith", name: "Bob Smith", city: "London" },
     { id: "ajones", name: "Alice Jones", city: "Paris" },
     { id: "dpeters", name: "Dora Peters", city: "New York" }];

let employees: Employee[] =
    [{ id: "bsmith", company: "Acme Co", dept: "Sales" },
     { id: "dpeters", company: "Acme Co", dept: "Development" }];

let dataItems: EmployedPerson[] = correlateData(people, employees);

function writePerson(per: Person): void {
    console.log('Person: ${per.id}, ${per.name}, ${per.city}');
}

function writeEmployee(emp: Employee): void {
    console.log('Employee: ${emp.id}, ${emp.company}, ${emp.dept}');
}

dataItems.forEach(item => {
    writePerson(item);
    writeEmployee(item);
});

```

Компилятор сопоставляет объект со структурой, гарантируя, что он содержит все свойства структуры, и не обращает внимания на дополнительные свойства (за исключением случаев определения объектного литерала, о чем говорилось ранее в текущей главе). Объекты, соответствующие типу `EmployedPerson`, могут быть переданы в функции `writePerson` и `writeEmployee`, так как они соответствуют типам, указанным в качестве параметров функции. Код в листинге 10.19 выдает следующий результат:

```

Person: bsmith, Bob Smith, London
Employee: bsmith, Acme Co, Sales
Person: ajones, Alice Jones, Paris
Employee: ajones, None, None
Person: dpeters, Dora Peters, New York
Employee: dpeters, Acme Co, Development

```

Хотя кажется очевидным, что тип пересечения совместим с каждым из составляющих его типов, это имеет важный эффект, когда типы в пересечении определяют свойства с одинаковыми именами: тип свойства в пересечении является пересечением типов отдельных свойств. Это утверждение трудно понять, поэтому в последующих разделах дается более развернутое объяснение.

Объединение свойств с одинаковым типом

Примером простейшей ситуации служит наличие свойств с одинаковым именем и типом, например свойство `id`, определенное типами `Person` и `Employee`, которые объединяются в пересечение без каких-либо изменений, как показано на рис. 10.8.

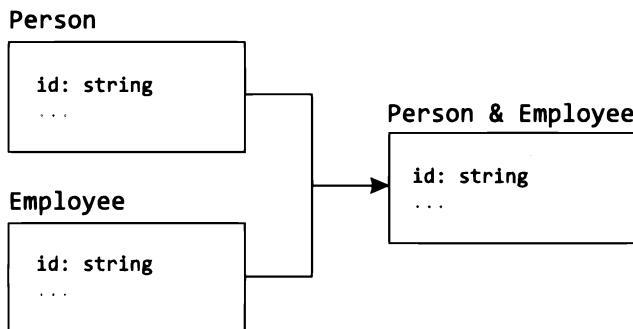


Рис. 10.8. Объединение свойств с одинаковым типом

В данной ситуации не возникает никаких проблем, поскольку любое значение, присваиваемое свойству `id`, будет строкой и будет соответствовать требованиям типов объектов и пересечений.

Объединение свойств с различными типами

Если имеются свойства с одинаковыми именами, но разными типами, компилятор сохраняет имя свойства, но пересекает их типы. Для демонстрации в листинге 10.20 удалены функции и добавлены свойства `contact` к типам `Person` и `Employee`.

Листинг 10.20. Добавление свойств с различными типами в файл `index.ts` из папки `src`

```

type Person = {
    id: string,
    name: string,
    city: string,
    contact: number
};

type Employee = {
    id: string,
    company: string,
    dept: string,
    contact: string
};

type EmployedPerson = Person & Employee;

let typeTest = ({} as EmployedPerson).contact;

```

Последнее утверждение в листинге 10.20 — это полезный прием, который позволяет узнать, какой тип компилятор присваивает свойству в пересечении, просмотрев файл декларации, созданный в папке `dist`, когда опция конфигурации компилятора `declaration` установлена в `true`. В этом объявлении используется утверждение типа, чтобы сообщить компилятору, что пустой объект соответствует типу `EmployedPerson`, и присвоить свойство `contact` переменной `typeTest`. После

сохранения изменений в файле `index.ts` компилятор скомпилирует код, и в файле `index.d.ts` в папке `dist` будет указан тип для свойства `contact` в пересечении.

```
declare let typeTest: never;
```

Поскольку между типами `string` и `number` нет пересечения, компилятор использовал для объединенного свойства тип `never`, как показано на рис. 10.9.

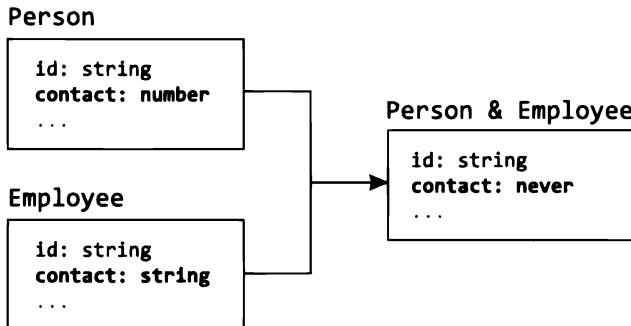


Рис. 10.9. Объединение свойств с различными типами

Создание пересечения типов — единственный способ, который позволяет компилятору объединить свойства. Однако в некоторых случаях это не даст полезного эффекта, так как не существует значений, которые можно было бы присвоить пересечению примитивных числового и строкового типов, как показано в листинге 10.21.

Листинг 10.21. Присвоение значений пересечениям в файле index.ts

```

type Person = {
    id: string,
    name: string,
    city: string,
    contact: number
};

type Employee = {
    id: string,
    company: string,
    dept: string,
    contact: string
};

type EmployedPerson = Person & Employee;

let typeTest = ({} as EmployedPerson).contact;

let person1: EmployedPerson = {
    id: "bsmith", name: "Bob Smith", city: "London",
    company: "Acme Co", dept: "Sales", contact: "Alice"
};

```

```
let person2: EmployedPerson = {
  id: "dpeters", name: "Dora Peters", city: "New York",
  company: "Acme Co", dept: "Development", contact: 6512346543
};
```

Для соответствия структуре объект должен присвоить свойству `contact` значение, но при этом возникают следующие ошибки:

```
src/index.ts(21,40): error TS2322: Type 'string' is not assignable to type
'never'.
```

```
src/index.ts(26,46): error TS2322: Type 'number' is not assignable to type
'never'.
```

Пересечение `string` и `number` является невозможным типом. Обойти эту проблему для примитивных типов невозможно, и единственное решение — это корректировка типов, используемых в пересечении, чтобы вместо примитивов использовались структуры типа, как показано в листинге 10.22.

ПРИМЕЧАНИЕ

Может показаться странным, что компилятор TypeScript допускает определение невозможных типов. Однако здесь следует понимать, что некоторые расширенные возможности TypeScript, которые будут рассмотрены в последующих главах, могут создавать ситуаций, в которых анализ всех вариантов становится чрезмерно сложным. В свете этого команда разработчиков Microsoft предпочла простоту, а не исчерпывающую проверку каждого невозможного типа.

Листинг 10.22. Использование структур типов в пересечении в файле index.ts

```
type Person = {
  id: string,
  name: string,
  city: string,
  contact: { phone: number }
};

type Employee = {
  id: string,
  company: string,
  dept: string,
  contact: { name: string }
};

type EmployedPerson = Person & Employee;

let typeTest = ({} as EmployedPerson).contact;

let person1: EmployedPerson = {
  id: "bsmith", name: "Bob Smith", city: "London",
  company: "Acme Co", dept: "Sales",
  contact: { name: "Alice" , phone: 6512346543 }
};
```

```
let person2: EmployedPerson = {
  id: "dpeters", name: "Dora Peters", city: "New York",
  company: "Acme Co", dept: "Development",
  contact: { name: "Alice" , phone: 6512346543 }
};
```

Компилятор обрабатывает объединение свойств аналогичным образом, но результатом пересечения служит структура, имеющая свойства `name` и `phone`, как показано на рис. 10.10.

Person

```
id: string
contact: { phone: number }
...
```

Employee

```
id: string
contact: { name: string }
...
```

Person & Employee

```
id: string
contact: { phone: number } & { name: string }
...
```

Рис. 10.10. Объединение свойств структур типов

Пересечением объекта со свойством `phone` и объекта со свойством `name` является объект со свойствами `phone` и `name`, что позволяет присваивать для `contact` значения, соответствующие типам `Person` и `Employee` и их пересечению.

Объединение методов

Если типы в пересечении определяют методы с одинаковыми именами, компилятор создаст функцию, сигнатурой которой является пересечение этих методов, как показано в листинге 10.23.

Листинг 10.23. Объединение методов в файле index.ts

```
type Person = {
  id: string,
  name: string,
  city: string,
  getContact(field: string): string
};

type Employee = {
  id: string,
  company: string,
  dept: string
  getContact(field: number): number
};
```

```

type EmployedPerson = Person & Employee;

let person: EmployedPerson = {
  id: "bsmith", name: "Bob Smith", city: "London",
  company: "Acme Co", dept: "Sales",
  getContact(field: string | number): any {
    return typeof field === "string" ? "Alice" : 6512346543;
  }
};

let typeTest = person.getContact;
let stringParamTypeTest = person.getContact("Alice");
let numberParamTypeTest = person.getContact(123);

console.log('Contact: ${person.getContact("Alice")}');
console.log('Contact: ${person.getContact(12)}');

```

Компилятор объединит функции, создав пересечение их сигнатур, что может привести к появлению невозможных типов или функций, которые не могут быть реализованы с пользой. В данном примере методы `getContact` в типах `Person` и `Employee` пересекаются, как показано на рис. 10.11.

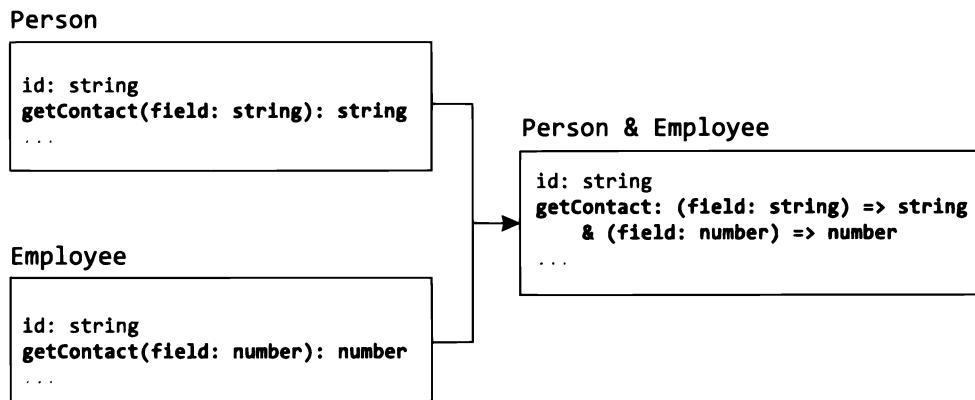


Рис. 10.11. Методы слияния

Бывает трудно разобраться с последствиями объединения методов в пересечении, но общий эффект схож с перегрузкой типов, описанной в главе 8. Лично я часто полагаюсь на файл декларации типов, чтобы убедиться, что я достиг желаемого пересечения. В листинге 10.23 есть три утверждения, которые помогают показать, как методы были объединены.

```

...
let typeTest = person.getContact;
let stringParamTypeTest = person.getContact("Alice");
let numberParamTypeTest = person.getContact(123);
...

```

После сохранения и компиляции файла `index.ts` файл `index.d.ts` в папке `dist` будет содержать утверждения, показывающие тип, который компилятор присвоил каждой из переменных:

```
declare let typeTest: ((field: string) => string)
& ((field: number) => number);
declare let stringParamTypeTest: string;
declare let numberParamTypeTest: number;
```

Первое утверждение показывает тип пересекаемого метода, а остальные — тип, возвращаемый при использовании строковых и числовых аргументов. Назначение файла `index.d.ts` подробно разъясняется в главе 15. Воспользоваться этой возможностью, чтобы увидеть типы, с которыми работает компилятор, часто бывает полезно.

Реализация пересекаемого метода должна сохранять совместимость с методами, входящими в пересечение. С параметрами обычно нет трудностей, и в листинге 10.23 используется объединение типов для создания метода, который может принимать значения типа `string` и `number`. С результатами метода дело обстоит сложнее, поскольку бывает нелегко подобрать тип, сохраняющий совместимость. Я считаю, что наиболее надежным подходом является использование типа `any` в качестве результата метода, а также применение строгих проверок типов для соответствия параметров и типов возвращаемых значений.

```
...
getContact(field: string | number): any {
    return typeof field === "string" ? "Alice" : 6512346543;
}
...
```

Хотя я стараюсь избегать использования типа `any`, в данном примере другого подходящего типа для того, чтобы объект `EmployedPerson` мог использоваться и как `Person`, и как `Employee`, просто нет. Код, приведенный в листинге 10.23, при компиляции и выполнении выдает следующий результат:

```
Contact: Alice
Contact: 6512346543
```

РЕЗЮМЕ

В данной главе мы рассмотрели, как TypeScript использует структуру объекта для проверки типов. Был объяснен процесс сопоставления структур, а также как эти структуры могут применяться для создания псевдонимов и образовывать объединения и пересечения.

- Комбинация типов, применяемых к свойствам и методам, формирует структуру типа объекта.
- Объекты соответствуют структуре типа, если они имеют свойства и методы с одинаковыми именами и типами.

- Структуры типа могут включать необязательные свойства, что позволяет объектам без этих свойств соответствовать типу.
- Объединения структур типов содержат только те свойства и методы, которые присущи всем типам, входящим в объединение. Любой член, не определенный всеми типами, исключается.
- Пересечения структур типов содержат все свойства и методы, определенные всеми типами в объединении, даже если они реализованы не всеми типами.
- Пересечения объединяют пересекающиеся свойства и методы на основе их типов.

В следующей главе вы познакомитесь с тем, как функции структур типов используются для обеспечения поддержки типов классов.

11

Работа с классами и интерфейсами

В этой главе

- ✓ Работа с типами для функций-конструкторов.
- ✓ Определение классов с помощью аннотаций типов.
- ✓ Ограничение доступа к членам класса с помощью элементов управления доступом.
- ✓ Упрощение классов за счет лаконичного синтаксиса конструктора.
- ✓ Создание свойств, доступных для изменения только внутри конструктора класса.
- ✓ Аксессоры и автоаксессоры.
- ✓ Концепция наследования классов.
- ✓ Интерфейсы и абстрактные классы.
- ✓ Динамическое создание свойств с использованием сигнатуры индекса.

В данной главе будут рассмотрены возможности TypeScript для работы с классами, а также функция интерфейса, предоставляющая альтернативный подход к описанию структуры объектов. В табл. 11.1 приведено краткое содержание главы.

В качестве краткой справки в табл. 11.2 перечислены параметры компилятора TypeScript, используемые в данной главе.

Таблица 11.1. Краткое содержание главы

Задача	Решение	Листинг
Последовательно создать объекты	Используйте функцию-конструктор или определите класс	4–6, 17–19
Запретить доступ к свойствам и методам	Используйте ключевые слова управления доступом TypeScript или приватные поля JavaScript	7–9
Запретить изменение свойств	Используйте ключевое слово <code>readonly</code>	10
Получить параметр конструктора и создать свойство экземпляра в один шаг	Воспользуйтесь лаконичным синтаксисом конструктора	11
Отделить доступ к данным от места их хранения	Используйте аксессоры или автоаксессоры	12–16
Определить частичную общую функциональность, которая будет унаследована подклассами	Определите абстрактный класс	20, 21
Определить структуру, которую могут реализовать классы	Определите интерфейс	12–27
Определить свойство динамически	Воспользуйтесь сигнатурой индекса	28–32

Таблица 11.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
<code>target</code>	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода
<code>outDir</code>	Задает каталог, в который будут помещены файлы JavaScript
<code>rootDir</code>	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
<code>declaration</code>	При включении этого параметра компилятор создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода. Более подробно эти файлы описаны в главе 15
<code>noUncheckedIndexedAccess</code>	Запрещает обращаться к свойствам через сигнатуру индекса, пока они не защищены от значений <code>undefined</code>

11.1. ПОДГОТОВКА К ИЗУЧЕНИЮ ГЛАВЫ

В текущей главе мы продолжим использовать проект `types`. Чтобы приступить к работе с материалом из этой главы, замените содержимое файла `index.ts` в папке `src` кодом из листинга 11.1.

Листинг 11.1. Замена содержимого файла index.ts из папки src

```
type Person = {
  id: string,
  name: string,
  city: string
};

let data: Person[] =
  [{ id: "bsmith", name: "Bob Smith", city: "London" },
   { id: "ajones", name: "Alice Jones", city: "Paris" },
   { id: "dpeters", name: "Dora Peters", city: "New York"}];

data.forEach(item => {
  console.log(`${item.id} ${item.name}, ${item.city}`);
});
```

Сбросьте конфигурацию компилятора, закомментировав опции конфигурации, как показано в листинге 11.2.

Листинг 11.2. Настройка компилятора в файле tsconfig.json из папки types

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    // "strictNullChecks": true
  }
}
```

В конфигурацию компилятора включен параметр `declaration`, который указывает компилятору создавать файлы декларации типов наряду с файлами JavaScript. Назначение этих файлов описано в главе 15, однако в текущей главе они будут использованы для пояснения работы компилятора с типами данных.

Откройте новое окно командной строки, перейдите в папку `types` и выполните команду из листинга 11.3 для запуска компилятора TypeScript, чтобы он автоматически выполнял код после его компиляции.

СОВЕТ

Проект примера для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 11.3. Запуск компилятора TypeScript

```
npm start
```

Компилятор скомпилирует проект, выполнит вывод, а затем перейдет в режим наблюдения, выдав следующий результат:

```
7:16:33 AM - Starting compilation in watch mode...
7:16:35 AM - Found 0 errors. Watching for file changes.
bsmith Bob Smith, London
ajones Alice Jones, Paris
dpeters Dora Peters, New York
```

11.2. ФУНКЦИИ-КОНСТРУКТОРЫ

Как мы уже знаем из главы 4, объекты могут быть созданы с помощью функций-конструкторов, предоставляя доступ к системе прототипов JavaScript. Функции-конструкторы можно использовать в коде TypeScript, но способ их поддержки не так интуитивно понятен и не столь элегантен, как работа с классами, о которых рассказывается далее в этой главе. В листинге 11.4 приведен пример кода, демонстрирующего функцию-конструктор.

Листинг 11.4. Использование функции-конструктора в файле index.ts из папки src

```
type Person = {
    id: string,
    name: string,
    city: string
};

let Employee = function(id: string, name: string, dept: string, city: string) {
    this.id = id;
    this.name = name;
    this.dept = dept;
    this.city = city;
};

Employee.prototype.writeDept = function() {
    console.log(`${this.name} works in ${this.dept}`);
};

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");

let data: (Person | Employee)[] =
    [{ id: "bsmith", name: "Bob Smith", city: "London" },
     { id: "ajones", name: "Alice Jones", city: "Paris" },
     { id: "dpeters", name: "Dora Peters", city: "New York" },
     salesEmployee];

data.forEach(item => {
    if (item instanceof Employee) {
        item.writeDept();
    } else {
        console.log(`${item.id} ${item.name}, ${item.city}`);
    }
});
```

Функция-конструктор Employee создает объекты со свойствами id, name, dept и city, а для прототипа Employee определен метод writeDept. Массив data обновляется и содержит объекты Person и Employee. Функция, переданная в метод forEach, использует оператор instanceof для сужения типа каждого объекта в массиве. Код в листинге 11.4 выдает следующие ошибки компилятора:

```
src/index.ts(20,21): error TS2749: 'Employee' refers to a value, but is
being used as a type here. Did you mean 'typeof Employee'?

src/index.ts(20,21): error TS4025: Exported variable 'data' has or is using
private name 'Employee'.

src/index.ts(28,14): error TS2339: Property 'writeDept' does not exist on type '{}'.  

```

TypeScript рассматривает функцию-конструктор `Employee` как обычную функцию и оценивает ее структуру, основываясь на типах параметров и возвращаемого значения. Когда функция `Employee` используется с ключевым словом `new`, компилятор присваивает объекту переменной `salesEmployee` тип `any`. В результате возникает ряд ошибок, поскольку компилятор пытается разобраться, как использовать функцию-конструктор.

Для решения этой проблемы необходимо предоставить компилятору дополнительную информацию о структурах используемых объектов. В листинге 11.5 добавлен псевдоним типа, описывающий объекты, создаваемые функцией-конструктором `Employee`.

Листинг 11.5. Добавление псевдонима типа в файл index.ts

```
type Person = {
    id: string,
    name: string,
    city: string
};

type Employee = {
    id: string,
    name: string,
    dept: string,
    city: string,
    writeDept: () => void
};

let Employee = function(id: string, name: string, dept: string,
    city: string) {
    this.id = id;
    this.name = name;
    this.dept = dept;
    this.city = city;
};
Employee.prototype.writeDept = function() {
    console.log(`${this.name} works in ${this.dept}`);
};

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");

let data: (Person | Employee)[] =
    [{ id: "bsmith", name: "Bob Smith", city: "London" },
     { id: "ajones", name: "Alice Jones", city: "Paris"}, 
     { id: "dpeters", name: "Dora Peters", city: "New York"}, 
     salesEmployee];

data.forEach(item => {
    if ("dept" in item) {
        item.writeDept();
    } else {
        console.log(`${item.id} ${item.name}, ${item.city}`);
    }
});
```

Компилятор TypeScript может не распознавать значения функции-конструктора, но он способен сопоставить создаваемые ею объекты по структуре. В листинге добавлена структура типа, соответствующая объектам, которые создаются функцией-конструктором, включая метод, доступ к которому осуществляется через прототип. В нашем примере для удобства структуре типа присвоен псевдоним, совпадающий с именем функции-конструктора. Однако это не обязательно, поскольку компилятор учитывает имена переменных и имена типов отдельно.

Обратите внимание, что в листинге 11.5 защита типа изменилась и теперь тип сужается за счет проверки свойства. Компилятор TypeScript не может использовать оператор `instanceof` в качестве защиты типа для объектов, создаваемых функцией-конструктором, поэтому был применен один из приемов, описанных в главе 10. В результате компилятор сопоставит структуру объектов, создаваемых функцией-конструктором `Employee`, со структурой, определяемой типом `Employee`, и сможет различать объекты по наличию свойства `dept`, выдавая при компиляции и выполнении кода следующий результат:

```
bsmith Bob Smith, London
ajones Alice Jones, Paris
dpeters Dora Peters, New York
Fidel Vega works in Sales
```

11.3. КЛАССЫ

TypeScript не имеет хорошей поддержки функций-конструкторов, но это связано с тем, что основное внимание уделяется поддержке классов. Это объясняется тем, что основной упор сделан на то, чтобы возможности, предоставляемые JavaScript, выглядели привычнее для программистов, привыкших к таким языкам, как C#. В листинге 11.6 фабричная функция заменяется классом.

Листинг 11.6. Использование класса в файле index.ts из папки src

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    id: string;
    name: string;
    dept: string;
    city: string;

    constructor(id: string, name: string, dept: string, city: string) {
        this.id = id;
        this.name = name;
        this.dept = dept;
        this.city = city;
    }
}
```

```

        writeDept() {
            console.log(`${this.name} works in ${this.dept}`);
        }
    }

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");

let data: (Person | Employee)[] =
    [{ id: "bsmith", name: "Bob Smith", city: "London" },
     { id: "ajones", name: "Alice Jones", city: "Paris" },
     { id: "dpeters", name: "Dora Peters", city: "New York" },
     salesEmployee];

data.forEach(item => {
    if (item instanceof Employee) {
        item.writeDept();
    } else {
        console.log(`${item.id} ${item.name}, ${item.city}`);
    }
});

```

Синтаксис класса TypeScript требует объявления свойств экземпляра и их типов. Это делает классы более объемными. Однако в этом есть свое преимущество: типы параметров конструктора могут отличаться от типов свойств экземпляра, которым они присваиваются. Объекты создаются из классов с помощью стандартного ключевого слова `new`, и компилятор понимает, что ключевое слово `instanceof` нужно для сужения типов при работе с классами.

Следует отметить, что TypeScript предоставляет мощные возможности для реализации классов. Класс в TypeScript может выглядеть иначе, чем стандартный класс JavaScript из главы 4. Однако важно понимать, что компилятор генерирует стандартные классы, которые во время выполнения зависят от функций-конструкторов и прототипов JavaScript. Вы можете увидеть класс, генерируемый из листинга 11.6, посмотрев на содержимое файла `index.js` в папке `dist`:

```

...
class Employee {
    id;
    name;
    dept;
    city;
    constructor(id, name, dept, city) {
        this.id = id;
        this.name = name;
        this.dept = dept;
        this.city = city;
    }
    writeDept() {
        console.log(`${this.name} works in ${this.dept}`);
    }
}
...

```

Когда вы начнете использовать более продвинутые возможности классов, может оказаться полезным проанализировать классы, создаваемые компилятором,

чтобы понять, как функции TypeScript преобразуются в чистый JavaScript. Код в листинге 11.6 при компиляции и выполнении выдает следующий результат:

```
bsmith Bob Smith, London
ajones Alice Jones, Paris
dpeters Dora Peters, New York
Fidel Vega works in Sales
```

11.3.1. Управление доступом с помощью ключевых слов

В JavaScript только недавно появилась поддержка приватных свойств и методов в классах с помощью символа `#`. TypeScript также поддерживает символ `#`, но при этом предоставляет более полный набор ключевых слов для управления доступом (табл. 11.3).

Таблица 11.3. Ключевые слова управления доступом в TypeScript

Название	Описание
<code>public</code>	Предоставляет свободный доступ к свойству или методу и используется по умолчанию, если ни одно ключевое слово не используется
<code>private</code>	Ограничивает всем доступ к свойству или методу указанного класса
<code>protected</code>	Ограничивает доступ к свойству или методу указанного класса всем, кроме его подклассов

TypeScript по умолчанию рассматривает свойства как `public`, если не указано ключевое слово. Однако для ясности кода рекомендуется явно указывать ключевое слово `public`. В листинге 11.7 применены ключевые слова к свойствам, определенным классом `Employee`.

Листинг 11.7. Применение ключевых слов управления доступом в файле index.ts

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    public id: string;
    public name: string;
    private dept: string;
    public city: string;

    constructor(id: string, name: string, dept: string, city: string) {
        this.id = id;
        this.name = name;
        this.dept = dept;
        this.city = city;
    }
}
```

```

    writeDept() {
      console.log(`${this.name} works in ${this.dept}`);
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");
console.log('Dept value: ${salesEmployee.dept}');

```

Ключевые слова управления доступом устанавливаются перед названием свойства, как показано на рис. 11.1.

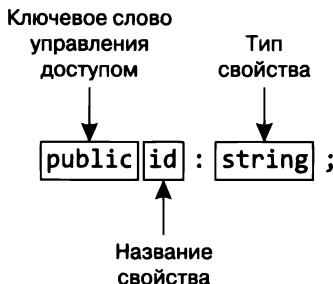


Рис. 11.1. Ключевое слово управления доступом

В листинге 11.7 ключевое слово `public` применено ко всем свойствам экземпляра, кроме `dept`, к которому было применено ключевое слово `private`. Это означает, что свойство `dept` доступно только внутри класса `Employee`. Если попытаться обратиться к свойству `dept` извне класса, компилятор выдаст ошибку:

```
src/index.ts(26,42): error TS2341: Property 'dept' is private and only accessible within class 'Employee'.
```

Единственный способ получить доступ к свойству `dept` — это использовать метод `writeDept` (листинг 11.8), который является частью класса `Employee` и разрешен ключевым словом `private`.

ВНИМАНИЕ

Ключевые слова управления доступом обеспечиваются компилятором TypeScript и не являются частью генерированного компилятором JavaScript-кода. Не полагайтесь на ключевое слово `private` или `protected` для защиты конфиденциальных данных, поскольку во время выполнения они будут доступны другим частям приложения.

Листинг 11.8. Использование метода в файле index.ts

```

type Person = {
  id: string,
  name: string,
  city: string
};

class Employee {
  public id: string;
  public name: string;
}

```

```
private dept: string;
public city: string;

constructor(id: string, name: string, dept: string, city: string) {
    this.id = id;
    this.name = name;
    this.dept = dept;
    this.city = city;
}

writeDept() {
    console.log(`${this.name} works in ${this.dept}`);
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");
salesEmployee.writeDept();
```

Код в листинге 11.8 при компиляции и выполнении выдает следующий результат:

```
Fidel Vega works in Sales
```

11.3.2. Приватные поля в JavaScript

TypeScript поддерживает стандартные для JavaScript приватные поля, введенные в спецификацию языка не так давно. Они работают так же, как ключевое слово `private`, как показано в листинге 11.9.

Листинг 11.9. Использование приватного поля в файле index.ts из папки src

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    public id: string;
    public name: string;
    #dept: string;
    public city: string;

    constructor(id: string, name: string, dept: string, city: string) {
        this.id = id;
        this.name = name;
        this.#dept = dept;
        this.city = city;
    }

    writeDept() {
        console.log(`${this.name} works in ${this.#dept}`);
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");
salesEmployee.writeDept();
```

Приватные поля обозначаются символом #, как показано на рис. 11.2.

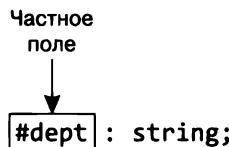


Рис. 11.2. Частное поле

Предфикс имени переменной dept ограничивает ее доступ к классу, который ее определяет. Символ # также необходим для получения или установки значения поля, например, так:

```
...
this.#dept = dept;
...
```

Главным преимуществом ключевого слова `private` в TypeScript является то, что символ # не удаляется в процессе компиляции, а значит, контроль доступа обеспечивается средой выполнения JavaScript. Как и большинство функций TypeScript, ключевое слово `private` не включается в JavaScript-код, генерируемый компилятором. Это означает, что контроль доступа в JavaScript-коде не обеспечивается. В листинге получен тот же результат, что и в предыдущем примере.

11.3.3. Определение свойств, доступных только для чтения

Ключевое слово `readonly` может быть использовано для создания свойств экземпляра, значение которых присваивается конструктором, но не может быть изменено иным образом, как показано в листинге 11.10.

Листинг 11.10. Создание свойства «только для чтения» в файле index.ts из папки src

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    public readonly id: string;
    public name: string;
    #dept: string;
    public city: string;

    constructor(id: string, name: string, dept: string, city: string) {
        this.id = id;
        this.name = name;
        this.#dept = dept;
        this.city = city;
    }
}
```

```

        writeDept() {
            console.log(`${this.name} works in ${this.#dept}`);
        }
    }

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");
salesEmployee.writeDept();
salesEmployee.id = "fidel";

```

Ключевое слово `readonly` должно идти после ключевого слова управления доступом, если оно было задействовано, как показано на рис. 11.3.



Рис. 11.3. Свойство «только для чтения»

Применение ключевого слова `readonly` к свойству `id` в листинге 11.11 означает, что значение, присвоенное конструктором, не может быть впоследствии изменено. Попытка присвоить новое значение свойству `id` вызовет следующую ошибку компилятора:

```
src/index.ts(27,15): error TS2540: Cannot assign to 'id' because it is a
read-only property.
```

11.3.4. Упрощение конструкторов классов

В классах JavaScript используются конструкторы для динамического создания свойств экземпляра, в то время как в TypeScript свойства должны быть определены явно. Подход TypeScript привычен большинству программистов, но он может быть громоздким и повторяющимся, особенно когда большинство параметров конструктора присваиваются свойствам, имеющим одно и то же имя. TypeScript поддерживает более лаконичный синтаксис для конструкторов, позволяющий избежать шаблона «определить и присвоить», как показано в листинге 11.11.

Листинг 11.11. Упрощение конструктора в файле index.ts из папки src

```

type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {

    constructor(public readonly id: string, public name: string,
                private dept: string, public city: string) {
        // не требуется никаких операторов
    }
}

```

```

    writeDept() {
      console.log(`${this.name} works in ${this.dept}`);
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");
salesEmployee.writeDept();
//salesEmployee.id = "fidel";

```

Для упрощения конструктора к параметрам применяются ключевые слова управления доступом, как показано на рис. 11.4.

```

Ключевое слово
управления доступом
↓
constructor( public readonly id: string, public name: string,
  private dept: string, public city: string) {

```

Рис. 11.4. Применение ключевых слов управления доступом к параметрам конструктора

Компилятор автоматически создает свойство экземпляра для каждого из аргументов конструктора, к которому было применено ключевое слово управления доступом, и присваивает ему значение параметра. Использование этих ключевых слов не изменяет способ вызова конструктора и требуется только для того, чтобы указать компилятору на необходимость создания соответствующих переменных экземпляра. Лаконичный синтаксис можно совмещать с обычными параметрами, а ключевое слово `readonly` будет автоматически применено к создаваемым компилятором свойствам экземпляра. Код, приведенный в листинге 11.11, выдает следующий результат:

```
Fidel Vega works in Sales
```

11.3.5. Определение аксессоров

Аксессоры — это функции `get` и `set`, которые используются для управления доступом к частному свойству класса. Они позволяют ввести дополнительную логику, отделяющую хранимое значение данных от способа его использования. Листинг 11.12 добавляет в класс функции `get` и `set`, обычно называемые *геттерами* и *сеттерами*.

Листинг 11.12. Добавление аксессоров в файл index.ts

```

type Person = {
  id: string,
  name: string,
  city: string
};

```

```

class Employee {
    private city: string;

    constructor(public readonly id: string, public name: string,
        private dept: string, city: string) {
        this.city = city;
    }

    writeDept() {
        console.log(`${this.name} works in ${this.dept}`);
    }

    get location() {
        return this.city;
    }

    set location(newCity) {
        this.city = newCity;
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");

salesEmployee.writeDept();
console.log('Location: ${salesEmployee.location}');
salesEmployee.location = "London";
console.log('Location: ${salesEmployee.location}');

```

Свойство, которым управляют с помощью аксессоров, называется *теневым полем* (backing field). В нашем примере это свойство `city`. Конструктор модифицирован таким образом, что значение, получаемое при создании объекта, присваивается частному свойству.

Ключевые слова `get` и `set` обозначают аксессоры и сопровождаются именем, которым в данном случае является `location`. Аксессор `get` — функция, возвращающая значение. Здесь возвращается значение свойства `city`. Сеттер — аксессор `set` — это функция, получающая новое значение `location`, которое сохраняется с помощью свойства `city`. Компилятор TypeScript выводит тип значения `location`, возвращаемого геттером, из теневого поля, и общий эффект получается таким, как если бы класс `Employee` определил свойство `location`. Когда мы читаем значение `location`, мы делаем это так, как если бы оно было свойством:

```

...
console.log('Location: ${salesEmployee.location}');
salesEmployee.location = "London";
...

```

Код в листинге 11.12 выдает следующий результат:

```

Fidel Vega works in Sales
Location: Paris
Location: London

```

Добавление косвенности в аксессоры

Аксессоры из листинга 11.12 управляют доступом к частному свойству. Геттер и сеттер представляют собой просто функции, а значит, могут вводить дополнительную логику. Из-за этого хранимое значение лишь косвенно связано со значением, предоставляемым через аксессоры, как показано в листинге 11.13.

Листинг 11.13. Добавление логики аксессоров в файл index.ts

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    private city: string;

    constructor(public readonly id: string, public name: string,
               private dept: string, city: string) {
        this.city = city;
    }

    writeDept() {
        console.log(`${this.name} works in ${this.dept}`);
    }

    get location() {
        switch (this.city) {
            case "Paris":
                return "France";
            case "London":
                return "UK";
            default:
                return this.city;
        }
    }

    set location(newCity) {
        this.city = newCity;
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");

salesEmployee.writeDept();
console.log('Location: ${salesEmployee.location}');
salesEmployee.location = "London";
console.log('Location: ${salesEmployee.location}'');
```

Геттер использует значение `city` для определения значения `location`. Эти два значения связаны, но связь между ними непрозрачна за пределами класса, где `location` используется так же, как свойство. Этот пример дает следующий результат, который показывает влияние логики геттера на значение `location`:

```
Fidel Vega works in Sales
Location: France
Location: UK
```

Аксессор get

Если аксессор `set` опущен, то в результате получается значение, которое ведет себя как свойство, доступное только для чтения, как показано в листинге 11.14.

Листинг 11.14. Удаление сеттера в файле index.ts

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    private city: string;

    constructor(public readonly id: string, public name: string,
               private dept: string, city: string) {
        this.city = city;
    }

    writeDept() {
        console.log(`${this.name} works in ${this.dept}`);
    }

    get location() {
        switch (this.city) {
            case "Paris":
                return "France";
            case "London":
                return "UK";
            default:
                return this.city;
        }
    }

    // set location(newCity) {
    //     this.city = newCity;
    // }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");

salesEmployee.writeDept();
console.log('Location: ${salesEmployee.location}');
// salesEmployee.location = "London";
// console.log('Location: ${salesEmployee.location}');
```

Значение `location` получается из свойства `city`, но присвоить новое значение `location` уже невозможно. В данном примере получен следующий результат:

```
Fidel Vega works in Sales
Location: France
```

Исключение теневого поля

Аксессоры обычно имеют теневое поле, но это не является обязательным условием. Геттеры и сеттеры могут использоваться для синтеза значений данных из других свойств класса, как показано в листинге 11.15.

Листинг 11.15. Аксессор без теневого поля в файле index.ts

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    private city: string;

    constructor(public readonly id: string, public name: string,
        private dept: string, city: string) {
        this.city = city;
    }

    writeDept() {
        console.log(`${this.name} works in ${this.dept}`);
    }

    get location() {
        switch (this.city) {
            case "Paris":
                return "France";
            case "London":
                return "UK";
            default:
                return this.city;
        }
    }

    get details() {
        return `${this.name}, ${this.dept}, ${this.location}`;
    }
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");

salesEmployee.writeDept();
console.log('Location: ${salesEmployee.location}');
console.log('Details: ${salesEmployee.details}');

    Геттер details не имеет собственного теневого поля, а возвращаемое им значение является производным от свойств name и dept, а также от аксессора location. Значение details по-прежнему считывается как обычное свойство, но возвращаемое значение создается с помощью композиции строк, как видно из примера:
Fidel Vega works in Sales
Location: France
Details: Fidel Vega, Sales, France

```

11.3.6. Автоаксессоры

Большинство аксессоров определяются с помощью теневых полей. Это настолько распространено явление, что была введена более лаконичная функция автоаксессора, как показано в листинге 11.16.

Листинг 11.16. Использование автоаксессора в файле index.ts из папки src

```
type Person = {
    id: string,
    name: string,
    city: string
};

class Employee {
    private city: string;

    constructor(public readonly id: string, public name: string,
        private dept: string, city: string) {
        this.city = city;
    }

    writeDept() {
        console.log(`${this.name} works in ${this.dept}`);
    }

    get location() {
        switch (this.city) {
            case "Paris":
                return "France";
            case "London":
                return "UK";
            default:
                return this.city;
        }
    }

    get details() {
        return `${this.name}, ${this.dept}, ${this.location}`;
    }

    accessor salary: number = 100_000;
}

let salesEmployee = new Employee("fvega", "Fidel Vega", "Sales", "Paris");

salesEmployee.writeDept();
console.log('Location: ${salesEmployee.location}');
console.log('Details: ${salesEmployee.details}');
console.log('Salary: ${salesEmployee.salary}');
```

Ключевое слово `accessor` обозначает автоаксессор, за которым следует имя и, опционально, начальное значение. Компилятор TypeScript определит тип аксессора по начальному значению, если оно указано, но можно использовать и аннотацию типа.

Автоаксессоры пока не включены в спецификацию языка JavaScript, поэтому компилятор TypeScript преобразует новое утверждение в листинге 11.16 в теневое поле с геттером и сеттером:

```
...
#salary_accessor_storage = 100000;
get salary() { return this.#salary_accessor_storage; }
set salary(value) { this.#salary_accessor_storage = value; }
...
```

Автоаксессор может быть заменен обычными аксессорами, если в дальнейшем потребуется дополнительная логика. В этом примере получается следующий вывод:

```
Fidel Vega works in Sales
Location: France
Details: Fidel Vega, Sales, France
Salary: 100000
```

11.3.7. Наследование классов

TypeScript опирается на стандартные возможности наследования классов, делая их более последовательными и привычными. Кроме того, он вводит несколько полезных дополнений для часто встречающихся задач и ограничивает некоторые характеристики JavaScript, которые могут вызвать проблемы. В листинге 11.17 псевдоним типа `Person` заменяется классом, предоставляющим те же возможности, и используется в качестве суперкласса (родительского класса) для `Employee`.

ПРИМЕЧАНИЕ

Хотя в примере показаны несколько классов в одном файле кода, общепринятой практикой является вынесение каждого класса в отдельный файл, что облегчает навигацию и понимание проекта. Более реалистичные примеры вы увидите в части III, где мы создадим серию веб-приложений.

Листинг 11.17. Добавление класса в файл index.ts

```
class Person {
    constructor(public id: string, public name: string, public city: string) { }

    class Employee extends Person {
        //private city: string;

        constructor(public readonly id: string, public name: string,
                   private dept: string, public city: string) {
            super(id, name, city);
        }

        writeDept() {
            console.log(`${this.name} works in ${this.dept}`);
        }

        // get location() {
        //     switch (this.city) {
```

```
//      case "Paris":
//      return "France";
//      case "London":
//      return "UK";
//      default:
//      return this.city;
//    }
// }

// get details() {
//   return `${this.name}, ${this.dept}, ${this.location}`;
// }

// accessor salary: number = 100_000;
}

// let salesEmployee = new Employee("fvega", "Fidel Vega",
//   "Sales", "Paris");

// salesEmployee.writeDept();
// console.log('Location: ${salesEmployee.location}');
// console.log('Details: ${salesEmployee.details}');
// console.log('Salary: ${salesEmployee.salary}');

let data = [new Person("bsmith", "Bob Smith", "London"),
  new Employee("fvega", "Fidel Vega", "Sales", "Paris")];

data.forEach(item => {
  console.log('Person: ${item.name}, ${item.city}');
  if (item instanceof Employee) {
    item.writeDept();
  }
});
```

При использовании ключевого слова `extends` язык TypeScript требует, чтобы конструктор родительского класса вызывался с помощью ключевого слова `super`, обеспечивая инициализацию его свойств. Код в листинге 11.17 выдает следующий результат:

```
Person: Bob Smith, London
Person: Fidel Vega, Paris
Fidel Vega works in Sales
```

Вывод типов для подклассов

Необходимо соблюдать осторожность, позволяя компилятору выводить типы из классов, поскольку, полагая, что компилятор имеет представление об иерархии классов, вы рискуете получить неожиданные результаты.

Массив данных в листинге 11.17 содержит объект `Person` и объект `Employee`. Если посмотреть на файл `index.d.ts` в папке `dist`, можно увидеть, что компилятор определил `Person[]` в качестве типа массива, как показано ниже:

```
...
declare let data: Person[];
...
...
```

Если вы знакомы с другими языками программирования, то вполне можете предположить, что компилятор понял, что `Employee` является подклассом класса `Person`, и что все объекты в массиве можно рассматривать как объекты `Person`. Однако в действительности компилятор создает объединение типов, содержащихся в массиве, которое будет иметь вид `Person | Employee`. Затем он определяет это как эквивалент типу `Person`, поскольку объединение представляет только общие для всех типов признаки. Важно помнить, что компилятор учитывает структуры объектов, даже если разработчик ориентируется на классы. Это кажется несущественным отличием, но оно имеет последствия при работе с объектами, имеющими сопутствующий суперкласс, как показано в листинге 11.18.

Листинг 11.18. Использование общего суперкласса в файле index.ts

```
class Person {

    constructor(public id: string, public name: string,
                public city: string) { }

}

class Employee extends Person {

    constructor(public readonly id: string, public name: string,
                private dept: string, public city: string) {
        super(id, name, city);
    }

    writeDept() {
        console.log(`${this.name} works in ${this.dept}`);
    }
}

class Customer extends Person {
    constructor(public readonly id: string, public name: string,
                public city: string, public creditLimit: number) {
        super(id, name, city);
    }
}

class Supplier extends Person {
    constructor(public readonly id: string, public name: string,
                public city: string, public companyName: string) {
        super(id, name, city);
    }
}

let data = [new Employee("fvega", "Fidel Vega", "Sales", "Paris"),
            new Customer("ajones", "Alice Jones", "London", 500)];

data.push(new Supplier("dpeters", "Dora Peters", "New York", "Acme"));

data.forEach(item => {
    console.log(`Person: ${item.name}, ${item.city}`);
    if (item instanceof Employee) {
        item.writeDept();
    } else if (item instanceof Customer) {
```

```

    console.log(`Customer ${item.name} has ${item.creditLimit} limit`);
} else if (item instanceof Supplier) {
    console.log(`Supplier ${item.name} works for ${item.companyName}`);
}
);

```

Этот пример не скомпилируется, поскольку компилятор TypeScript определил тип массива `data` на основе типов содержащихся в нем объектов и не отразил общий суперкласс. Вот утверждение из файла `index.d.ts` в папке `dist`, которое показывает тип, выведенный компилятором:

```

...
declare let data: (Employee | Customer)[];
...

```

Массив может содержать только объекты `Employee` или `Customer`, а ошибки выдаются из-за добавления объекта `Supplier`. Для решения этой проблемы можно использовать аннотацию типа, чтобы указать компилятору, что массив может содержать объекты `Product`, как показано в листинге 11.19.

Листинг 11.19. Использование аннотации типа в файле index.ts

```

...
let data: Person[] = [new Employee("fvega", "Fidel Vega", "Sales",
    "Paris"), new Customer("ajones", "Alice Jones", "London", 500)];
data.push(new Supplier("dpeters", "Dora Peters", "New York", "Acme"));
...

```

Компилятор позволит массиву данных хранить объекты `Product` и объекты, созданные на основе его подклассов. Код в листинге 11.19 выдает следующий результат:

```

Person: Fidel Vega, Paris
Fidel Vega works in Sales
Person: Alice Jones, London
Customer Alice Jones has 500 limit
Person: Dora Peters, New York
Supplier Dora Peters works for Acme

```

11.3.8. Абстрактные классы

Абстрактные классы не могут создать экземпляры напрямую, а служат для описания общей функциональности, которую должны реализовать подклассы. Это обязывает подклассы следовать определенной структуре, но также предоставляет возможность реализовать специфические для каждого класса методы, как показано в листинге 11.20.

Листинг 11.20. Определение абстрактного класса в файле index.ts

```

abstract class Person {

    constructor(public id: string, public name: string, public city: string) { }

    getDetails(): string {
        return `${this.name}, ${this.getSpecificDetails()}`;
    }
}

```

```

    abstract getSpecificDetails(): string;
}

class Employee extends Person {

    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        super(id, name, city);
    }

    getSpecificDetails() {
        return 'works in ${this.dept}';
    }
}

class Customer extends Person {

    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number) {
        super(id, name, city);
    }

    getSpecificDetails() {
        return 'has ${this.creditLimit} limit';
    }
}

class Supplier extends Person {

    constructor(public readonly id: string, public name: string,
        public city: string, public companyName: string) {
        super(id, name, city);
    }

    getSpecificDetails() {
        return 'works for ${this.companyName}';
    }
}

let data: Person[] = [new Employee("fvega", "Fidel Vega", "Sales",
    "Paris"), new Customer("ajones", "Alice Jones", "London", 500)];
data.push(new Supplier("dpeters", "Dora Peters", "New York", "Acme"));

data.forEach(item => console.log(item.getDetails()));

```

Абстрактные классы создаются с помощью ключевого слова `abstract` перед ключевым словом `class`, как показано на рис. 11.5.

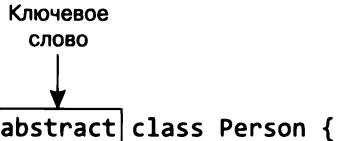


Рис. 11.5. Определение абстрактного класса

Ключевое слово `abstract` применяется также к отдельным методам, которые определяются без тела, как показано на рис. 11.6.



Рис. 11.6. Определение абстрактного метода

Когда класс расширяет абстрактный класс, он должен реализовать все абстрактные методы. В нашем примере абстрактный класс `Person` определяет абстрактный метод `getSpecificDetails`, который должен быть реализован классами `Employee`, `Customer` и `Supplier`. Класс `Person` также определяет обычный метод `getDetails`, который вызывает абстрактный метод и использует его результат.

Объекты, созданные из классов, унаследованных от класса `abstract`, могут использоваться через тип абстрактного класса, то есть объекты `Employee`, `Customer` и `Supplier` могут храниться в массиве `Person`, но при этом использовать только свойства и методы, определенные классом `Person`, если объекты не сужены до более конкретного типа. Код в листинге 11.20 выдает следующий результат:

```

Fidel Vega, works in Sales
Alice Jones, has 500 limit
Dora Peters, works for Acme

```

Защита типа абстрактного класса

Абстрактные классы реализуются как обычные классы в JavaScript, которые генерирует компилятор TypeScript. Недостаток этого подхода заключается в том, что именно компилятор TypeScript не позволяет инстанцировать абстрактные классы, и это не отражается в сформированном JavaScript-коде, что потенциально позволяет создавать объекты на основе абстрактного класса. Тем не менее подобный подход означает, что ключевое слово `instanceof` можно использовать для сужения типов, как показано в листинге 11.21.

Листинг 11.21. Защита типа абстрактного класса в файле index.ts

```

abstract class Person {

    constructor(public id: string, public name: string,
               public city: string) { }

    getDetails(): string {
        return `${this.name}, ${this.getSpecificDetails()}`;
    }

    abstract getSpecificDetails(): string;
}

class Employee extends Person {

```

```

constructor(public readonly id: string, public name: string,
            private dept: string, public city: string) {
    super(id, name, city);
}

getSpecificDetails() {
    return `works in ${this.dept}`;
}
}

class Customer {

    constructor(public readonly id: string, public name: string,
                public city: string, public creditLimit: number) {
    }
}

let data: (Person | Customer)[] = [
    new Employee("fvega", "Fidel Vega", "Sales", "Paris"),
    new Customer("ajones", "Alice Jones", "London", 500)];

```

data.forEach(item => {
 if (item instanceof Person) {
 console.log(item.getDetails());
 } else {
 console.log(`Customer: \${item.name}`);
 }
});

В данном листинге класс `Employee` расширяет абстрактный класс `Person`, а класс `Customer` — нет. Оператор `instanceof` дает возможность идентифицировать любой объект, созданный из класса, расширяющего абстрактный класс. Это позволяет сузить объединение типов `Person | Customer`, используемое в качестве типа для массива. Код в листинге 11.21 выдает следующий результат:

```
Fidel Vega, works in Sales
Customer: Alice Jones
```

11.4. ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСОВ

Интерфейсы используются для определения структуры объекта, которой должен соответствовать класс, реализующий данный интерфейс, как показано в листинге 11.22.

ПРИМЕЧАНИЕ

Интерфейсы имеют сходное назначение с типами структур, описанными в главе 10, и в последних версиях TypeScript различия между этими двумя функциями стирались до такой степени, что их часто можно использовать как взаимозаменяемые для достижения одного и того же эффекта, особенно при работе с простыми типами. Тем не менее интерфейсы обладают рядом полезных свойств и обеспечивают опыт разработки, более схожий с такими языками, как C#.

Листинг 11.22. Использование интерфейса в файле index.ts

```

interface Person {
    name: string;
    getDetails(): string;
}

class Employee implements Person {

    constructor(public readonly id: string, public name: string,
                private dept: string, public city: string) {
        // не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} works in ${this.dept}`;
    }
}

class Customer implements Person {

    constructor(public readonly id: string, public name: string,
                public city: string, public creditLimit: number) {
        // не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} has ${this.creditLimit} limit`;
    }
}

let data: Person[] = [
    new Employee("fvega", "Fidel Vega", "Sales", "Paris"),
    new Customer("ajones", "Alice Jones", "London", 500)];
data.forEach(item => console.log(item.getDetails()));

```

Интерфейсы объявляются ключевым словом `interface` и содержат набор свойств и методов, которыми должен обладать класс, чтобы соответствовать интерфейсу, как показано на рис. 11.7.

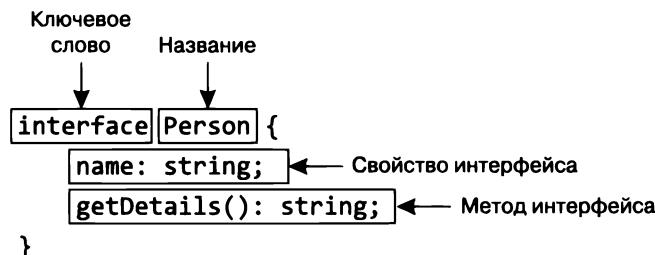
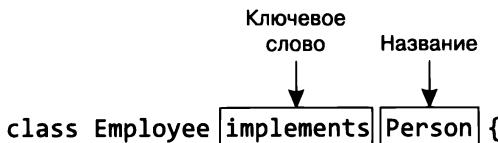


Рис. 11.7. Определение интерфейса

В отличие от абстрактных классов, интерфейсы не реализуют методы и не определяют конструктор, а только определяют структуру. Интерфейсы реализуются классами с помощью ключевого слова `implements`, как показано на рис. 11.8.

**Рис. 11.8.** Реализация интерфейса

Интерфейс `Person` определяет свойство `name` и метод `getDetails`, поэтому классы `Employee` и `Customer` должны определять то же свойство и метод. Эти классы могут иметь дополнительные свойства и методы, но они должны соответствовать интерфейсу, предоставляя свойства `name` и `getDetails`. Интерфейс можно использовать в аннотациях типов, как массив в примере.

```

...
let data: Person[] = [
    new Employee("fvega", "Fidel Vega", "Sales", "Paris"),
    new Customer("ajones", "Alice Jones", "London", 500)];
...

```

Массив данных может содержать любой объект, созданный на основе класса, реализующего массив `Product`, хотя функция, передаваемая в метод `forEach`, в состоянии обращаться только к тем свойствам и методам, которые определены интерфейсом, если только объекты не сужены до более конкретного типа. Код в листинге 11.22 выдает следующий результат:

```

Fidel Vega works in Sales
Alice Jones has 500 limit

```

ОБЪЕДИНЕНИЕ ДЕКЛАРАЦИЙ ИНТЕРФЕЙСОВ

Интерфейсы могут быть определены в нескольких декларациях `interface`, которые объединяются компилятором в один интерфейс. Эта особенность кажется необычной, и пока что я не нашел ей практического применения в своих проектах. Все декларации должны находиться в одном файле кода и либо быть экспортированы (определенны ключевым словом `export`), либо определены локально (без ключевого слова `export`).

11.4.1. Реализация нескольких интерфейсов

Класс может реализовывать более одного интерфейса, то есть он должен определять методы и свойства, установленные всеми этими интерфейсами, как показано в листинге 11.23.

Листинг 11.23. Реализация нескольких интерфейсов в файле index.ts

```

interface Person {
    name: string;
    getDetails(): string;
}

```

```

interface DogOwner {
    dogName: string;
    getDogDetails(): string;
}

class Employee implements Person {
    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        // не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} works in ${this.dept}`;
    }
}

class Customer implements Person, DogOwner {
    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number, public dogName ) {
        // не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} has ${this.creditLimit} limit`;
    }

    getDogDetails() {
        return `${this.name} has a dog named ${this.dogName}`;
    }
}

let alice = new Customer("ajones", "Alice Jones", "London", 500, "Fido");

let dogOwners: DogOwner[] = [alice];
dogOwners.forEach(item => console.log(item.getDogDetails()));

let data: Person[] = [new Employee("fvega", "Fidel Vega", "Sales", "Paris"), alice];
data.forEach(item => console.log(item.getDetails()));

```

Интерфейсы перечисляются после ключевого слова `implements` через запятую. В примере из листинга класс `Customer` реализует интерфейсы `Person` и `DogOwner`. Это значит, что объект `Person`, присвоенный переменной с именем `alice`, может быть добавлен в массивы, типизированные для объектов `Person` и `DogOwner`. Код в листинге 11.23 выдает следующий результат:

```

Alice Jones has a dog named Fido
Fidel Vega works in Sales
Alice Jones has 500 limit

```

ПРИМЕЧАНИЕ

Класс может реализовать несколько интерфейсов только в том случае, если в них нет пересекающихся свойств с конфликтующими типами. Например, если интерфейс `Person` определяет строковое свойство `id`, а интерфейс `DogOwner` — числовое свойство с тем же именем, то класс `Customer` не сможет реализовать оба интерфейса, поскольку свойству `id` нельзя присвоить значение, представляющее оба типа.

11.4.2. Расширение интерфейсов

Интерфейсы, как и классы, могут быть расширены. При этом используется тот же базовый подход. В результате получается интерфейс, содержащий свойства и методы, унаследованные от его родительских интерфейсов, а также все новые функции, которые определены, как показано в листинге 11.24.

Листинг 11.24. Расширение интерфейса в файле index.ts

```
interface Person {
    name: string;
    getDetails(): string;
}

interface DogOwner extends Person {
    dogName: string;
    getDogDetails(): string;
}

class Employee implements Person {

    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        // Не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} works in ${this.dept}`;
    }
}

class Customer implements DogOwner {

    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number,
        public dogName ) {
        // Не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} has ${this.creditLimit} limit`;
    }

    getDogDetails() {
        return `${this.name} has a dog named ${this.dogName}`;
    }
}

let alice = new Customer("ajones", "Alice Jones", "London", 500, "Fido");

let dogOwners: DogOwner[] = [alice];
dogOwners.forEach(item => console.log(item.getDogDetails()));

let data: Person[] = [new Employee("fvega", "Fidel Vega", "Sales",
    "Paris"), alice];
data.forEach(item => console.log(item.getDetails()));
```

Ключевое слово `extend` используется для расширения интерфейса. В листинге выше интерфейс `DogOwner` расширяет интерфейс `Person`, а это означает, что классы, реализующие `DogOwner`, должны определять свойства и методы обоих интерфейсов. Объекты, созданные на основе класса `Customer`, допускается рассматривать как объекты и `DogOwner`, и `Person`, поскольку они всегда определяют структуры, требуемые каждым из интерфейсов. Код, приведенный в листинге 11.24, выдает следующий результат:

```
Alice Jones has a dog named Fido
Fidel Vega works in Sales
Alice Jones has 500 limit
```

11.4.3. Необязательные свойства и методы интерфейса

Добавление необязательного свойства в интерфейс позволяет классам, реализующим этот интерфейс, предоставлять это свойство, не делая его обязательным, как показано в листинге 11.25.

Листинг 11.25. Добавление необязательного свойства в файл index.ts

```
interface Person {
    name: string;
    getDetails(): string;

    dogName?: string;
    getDogDetails?(): string;
}

class Employee implements Person {

    constructor(public readonly id: string, public name: string,
        private dept: string, public city: string) {
        // не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} works in ${this.dept}`;
    }
}

class Customer implements Person {

    constructor(public readonly id: string, public name: string,
        public city: string, public creditLimit: number,
        public dogName) {
        // не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} has ${this.creditLimit} limit`;
    }
}
```

```

    getDogDetails() {
        return `${this.name} has a dog named ${this.dogName}`;
    }
}

let alice = new Customer("ajones", "Alice Jones", "London", 500, "Fido");
let data: Person[] = [new Employee("fvega", "Fidel Vega", "Sales",
    "Paris"), alice];
data.forEach(item => {
    console.log(item.getDetails());
    if (item.getDogDetails) {
        console.log(item.getDogDetails());
    }
});

```

Объявление необязательного свойства в интерфейсе производится с помощью символа ? после имени, как показано на рис. 11.9.

```

interface Person {
    name: string;
    getDetails(): string;
    dogName?: string; ← Необязательное свойство
    getDogDetails?(): string; ← Необязательный метод
}

```

Рис. 11.9. Определение необязательных членов интерфейса

Необязательные функции интерфейса могут определяться через тип интерфейса, не вызывая ошибок компилятора, однако необходимо убедиться, что вы не получите значение undefined, поскольку существует риск, что объекты созданы из классов, не реализующих эти методы:

```

...
data.forEach(item => {
    console.log(item.getDetails());
    if (item.getDogDetails) {
        console.log(item.getDogDetails());
    }
});
...

```

Только один из типов в листинге 11.25, реализующий интерфейс Person, определяет метод getDogDetails. Доступ к этому методу можно получить через тип Person, не сужаясь до конкретного класса. Тем не менее он может быть не определен, поэтому мы используем приведение типа в условном выражении, чтобы метод вызывался только на объектах, которые его определили. Код в листинге 11.25 выдает следующий результат:

```

Fidel Vega works in Sales
Alice Jones has 500 limit
Alice Jones has a dog named Fido

```

11.4.4. Определение реализации абстрактного интерфейса

Абстрактные классы могут быть использованы для реализации некоторых или всех функций, описываемых интерфейсом, как показано в листинге 11.26. Это позволяет сократить дублирование кода, когда некоторые классы используют один и тот же код для реализации функциональности.

Листинг 11.26. Создание абстрактной реализации в файле index.ts

```
interface Person {
    name: string;
    getDetails(): string;

    dogName?: string;
    getDogDetails?(): string;
}

abstract class AbstractDogOwner implements Person {

    abstract name: string;
    abstract dogName?: string;

    abstract getDetails();

    getDogDetails() {
        if (this.dogName) {
            return `${this.name} has a dog called ${this.dogName}`;
        }
    }
}

class DogOwningCustomer extends AbstractDogOwner {

    constructor(public readonly id: string, public name: string,
                public city: string, public creditLimit: number,
                public dogName) {
        super();
    }

    getDetails() {
        return `${this.name} has ${this.creditLimit} limit`;
    }
}

let alice = new DogOwningCustomer("ajones", "Alice Jones", "London",
    500, "Fido");
if (alice.getDogDetails) {
    console.log(alice.getDogDetails());
}
```

AbstractDogOwner обеспечивает частичную реализацию интерфейса Person, но объявляет нереализованные функции интерфейса абстрактными (`abstract`), что

вынуждает подклассы реализовывать их. Существует один подкласс, который расширяет `AbstractDogOwner` и наследует метод `getDogDetails` от абстрактного класса. Код в листинге 11.26 выдает следующий результат:

```
Alice Jones has a dog called Fido
```

11.4.5. Защита типа интерфейса

В JavaScript не существует эквивалента интерфейсов, и в JavaScript-код, генерируемый компилятором TypeScript, не включаются никакие детали интерфейсов. Это означает, что невозможно использовать ключевое слово `instanceof` для сужения типов интерфейсов, а защита типов может быть выполнена только путем проверки наличия одного или нескольких свойств, определенных интерфейсом, как показано в листинге 11.27.

Листинг 11.27. Защита типа интерфейса в файле index.ts

```
interface Person {
    name: string;
    getDetails(): string;
}

interface Product {
    name: string;
    price: number;
}

class Employee implements Person {

    constructor(public name: string, public company: string) {
        // не требуется никаких операторов
    }

    getDetails() {
        return `${this.name} works for ${this.company}`;
    }
}

class SportsProduct implements Product {
    constructor(public name: string, public category: string,
               public price: number) {
        // не требуется никаких операторов
    }
}

let data: (Person | Product)[] = [new Employee("Bob Smith", "Acme"),
    new SportsProduct("Running Shoes", "Running", 90.50),
    new Employee("Dora Peters", "BigCo")];

data.forEach(item => {
    if ("getDetails" in item) {
        console.log('Person: ${item.getDetails()}');
    } else {
        console.log('Product: ${item.name}, ${item.price}');
    }
});
```

В данном листинге наличие свойства `getDetails` используется для идентификации объектов, реализующих интерфейс `Person`, что позволяет сузить содержимое массива `data` до типа `Person` или `Product`. В листинге 11.27 получен следующий результат:

```
Person: Bob Smith works for Acme
Product: Running Shoes, 90.5
Person: Dora Peters works for BigCo
```

11.5. ДИНАМИЧЕСКОЕ СОЗДАНИЕ СВОЙСТВ

Компилятор TypeScript позволяет присваивать значения только тем свойствам, которые являются частью типа объекта, то есть интерфейсы и классы должны определять все необходимые приложению свойства.

В JavaScript же можно создавать новые свойства объектов, просто присваивая значение неиспользуемому имени свойства. Функция *сигнатуры индекса* в TypeScript соединяет эти две модели, предоставляя возможность динамически определять свойства, сохраняя при этом безопасность типов, как показано в листинге 11.28.

Листинг 11.28. Определение сигнатуры индекса в файле index.ts

```
interface Product {
    name: string;
    price: number;
}

class SportsProduct implements Product {
    constructor(public name: string, public category: string,
                public price: number) {
        // не требуется никаких операторов
    }
}

class ProductGroup {
    constructor(...initialProducts: [string, Product][]) {
        initialProducts.forEach(p => this[p[0]] = p[1]);
    }

    [propertyName: string]: Product;
}

let group = new ProductGroup(["shoes", new SportsProduct("Shoes",
    "Running", 90.50)]);
group.hat = new SportsProduct("Hat", "Skiing", 20);
Object.keys(group).forEach(k => console.log('Property Name: ${k}'));
```

Класс `ProductGroup` получает через свой конструктор массив кортежей `[string, Product]`, каждый из которых необходим для создания свойства, использующего в качестве имени строковое значение, а в качестве значения — `Product`. Компилятор разрешит конструктору создать свойство и присвоить ему тип `any`, если только не включены параметры компилятора `noImplicitAny` или `strict`. В этом случае будет выдана ошибка.

Чтобы динамически создавать свойства вне конструктора (и избежать ошибок компилятора `noImplicitAny`), классы могут определять сигнатуру индекса. В ней используются квадратные скобки для указания типа ключей свойств, за которыми следует аннотация типа, ограничивающая типы, которые могут быть использованы для создания динамических свойств (рис. 11.10).

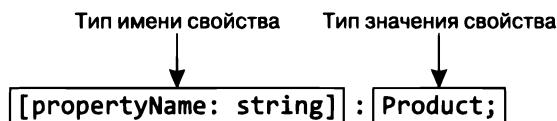


Рис. 11.10. Сигнатура индекса

Тип имени свойства предполагает только `string` или `number`, в то время как тип значения свойства может быть любым. Сигнатура индекса на рисунке указывает компилятору разрешить динамические свойства, которые используют значения типа `string` для имен и которым присваиваются значения `Product`, как, например, этому свойству:

```

...
group.hat = new SportsProduct("Hat", "Skiing", 20);
...

```

Этот оператор создает свойство с именем `hat`. Код в листинге 11.28 выводит следующий результат, отображая имена свойств, созданных конструктором и последующим оператором:

```

Property Name: shoes
Property Name: hat

```

11.5.1. Включение проверки значения индекса

Одна из потенциальных ошибок при использовании сигнатур индекса заключается в том, что компилятор TypeScript рассчитывает, что вы будете обращаться только к существующим свойствам. Это не согласуется с более широким подходом, применяемым в TypeScript, который предполагает, что все элементы должны быть явно определены, чтобы их можно было проверить. В листинге 11.29 мы обращаемся к несуществующему свойству с помощью сигнатуры индекса.

Листинг 11.29. Доступ к несуществующему свойству в файле `index.ts`

```

interface Product {
    name: string;
    price: number;
}

class SportsProduct implements Product {
    constructor(public name: string, public category: string,
                public price: number) {
        // не требуется никаких операторов
    }
}

```

```

class ProductGroup {
    constructor(...initialProducts: [string, Product][]) {
        initialProducts.forEach(p => this[p[0]] = p[1]);
    }

    [propertyName: string]: Product;
}

let group = new ProductGroup(["shoes", new SportsProduct("Shoes",
    "Running", 90.50)]);
group.hat = new SportsProduct("Hat", "Skiing", 20);

let total = group.hat.price + group.boots.price;
console.log('Total: ${total}');

```

Оператор, присваивающий значение `total`, использует сигнатуру индекса для доступа к свойствам `hat` и `boots`. Свойство `boots` не было создано, но код все равно компилируется, и при его выполнении возникает ошибка.

```
let total = group.hat.price + group.boots.price;
```

TypeError: Cannot read properties of undefined (reading 'price')

Чтобы настроить компилятор на проверку доступа к сигнатурам индекса, установите опции конфигурации `noUncheckedIndexedAccess` и `strictNullChecks` в значение `true`, как показано в листинге 11.30.

Листинг 11.30. Настройка компилятора в файле tsconfig.json из папки types

```
{
    "compilerOptions": {
        "target": "ES2022",
        "outDir": "./dist",
        "rootDir": "./src",
        "declaration": true,
        "strictNullChecks": true,
        "noUncheckedIndexedAccess": true
    }
}
```

После сохранения изменений код будет перекомпилирован. На этот раз компилятор TypeScript выдаст следующую ошибку:

`src/index.ts(25,31): error TS18048: 'group.boots' is possibly 'undefined'.`

Для ее предотвращения необходимо убедиться, что нужное свойство существует, прежде чем пытаться использовать его значение для защиты от значений типа `undefined` (листинг 11.31).

Листинг 11.31. Проверка свойства в файле index.ts из папки src

```

interface Product {
    name: string;
    price: number;
}

```

```

class SportsProduct implements Product {
    constructor(public name: string, public category: string, public price: number) {
        // не требуется никаких операторов
    }
}

class ProductGroup {
    constructor(...initialProducts: [string, Product][]) {
        initialProducts.forEach(p => this[p[0]] = p[1]);
    }

    [propertyName: string]: Product;
}

let group = new ProductGroup(["shoes", new SportsProduct("Shoes", "Running", 90.50)]);
group.hat = new SportsProduct("Hat", "Skiing", 20);

if (group.hat && group.boots) {
    let total = group.hat.price + group.boots.price;
    console.log('Total: ${total}');
}

```

Выражение `if` гарантирует, что свойство `boots` не будет использоваться, если оно имеет тип `undefined`. Альтернативный подход заключается в применении optional chaining и оператора `??` для предоставления резервного значения, как показано в листинге 11.32.

Листинг 11.32. Использование резервного значения в файле index.ts из папки src

```

interface Product {
    name: string;
    price: number;
}

class SportsProduct implements Product {
    constructor(public name: string, public category: string, public price: number) {
        // не требуется никаких операторов
    }
}

class ProductGroup {
    constructor(...initialProducts: [string, Product][]) {
        initialProducts.forEach(p => this[p[0]] = p[1]);
    }

    [propertyName: string]: Product;
}

let group = new ProductGroup(["shoes", new SportsProduct("Shoes", "Running", 90.50)]);
group.hat = new SportsProduct("Hat", "Skiing", 20);

let total = group.hat.price + (group.boots?.price ?? 0);
console.log('Total: ${total}');

```

Этот код выдает следующий результат:

```
Total: 20
```

РЕЗЮМЕ

В этой главе вы узнали, как TypeScript расширяет возможности классов JavaScript, обеспечивая поддержку лаконичного синтаксиса конструкторов, абстрактных классов и ключевых слов управления доступом. Вы также познакомились с функцией интерфейса, которая реализуется компилятором и предоставляет альтернативный способ описания структуры объектов, чтобы классы могли им соответствовать.

- TypeScript поддерживает статические типы в функциях-конструкторах, однако работать с ними не так удобно, как с классами.
- В TypeScript хорошо реализована поддержка классов и компилятор выводит типы непосредственно из определения класса.
- TypeScript предлагает простой синтаксис для конструкторов классов, который создает публичные свойства для каждого аргумента публичного конструктора.
- TypeScript поддерживает синтаксис JavaScript для приватных членов класса, а также ключевые слова `public`, `private` и `protected`, которые обеспечивают более детальный контроль доступа.
- Ключевое слово `readonly` позволяет создавать свойства, которые можно изменять только в конструкторе класса, в котором они определены.
- Наследование классов осуществляется путем создания объединения типов, при этом в новый тип включаются только общие черты.
- Абстрактные классы могут использоваться для создания базовых реализаций функций, которые затем наследуются подклассами.
- Интерфейсы описывают свойства и методы, которые должен реализовать класс, чтобы соответствовать определенному типу.
- TypeScript поддерживает функцию JavaScript для динамического создания свойств объектов, которая обрабатывается через сигнатуры индексов для проверки типов.

В следующей главе мы поговорим о поддержке обобщенных типов TypeScript.

Следующий раздел посвящен обобщенным типам, которые позволяют использовать один и тот же код для работы с различными типами. В TypeScript обобщенные типы называются дженериками. В главе 13 мы познакомимся с основами дженериков, а в данной главе — с более продвинутыми возможностями.

Обобщенные типы

В этой главе

- ✓ Использование параметров обобщенных типов в качестве заполнителей типов.
- ✓ Создание экземпляров классов с аргументами обобщенного типа.
- ✓ Ограничение параметров обобщенных типов.
- ✓ Защита обобщенных типов с помощью функций-предикатов.
- ✓ Определение интерфейсов с параметрами обобщенного типа.

Обобщенные типы (*generic types*), или дженерики, представляют собой специальные заполнители для типов, которые заменяются конкретными типами при использовании класса или функции. Это позволяет писать типобезопасный код, способный работать с различными типами, например с классами коллекций. Этую концепцию легче продемонстрировать, чем объяснить, поэтому мы начнем данную главу с примера проблемы, которую решают обобщенные типы, а затем рассмотрим основные способы использования различных обобщенных типов. В главе 13 будут представлены расширенные возможности обобщенных типов, которые предоставляет TypeScript.

В табл. 12.1 приведено краткое содержание главы.

В качестве краткой справки в табл. 12.2 перечислены параметры компилятора TypeScript, используемые в данной главе.

Таблица 12.1. Краткое содержание главы

Задача	Решение	Листинг
Определить класс или функцию, которые могут безопасно работать с другими различными типами	Задайте параметр обобщенного типа	6–8, 20, 21
Определить тип для параметра обобщенного типа	Используйте аргумент обобщенного типа при создании экземпляра класса или вызове функции	9–14
Расширить обобщенный класс	Создайте класс, который передает, ограничивает или исправляет параметр обобщенного типа, унаследованный от родительского класса	15–17
Задать защиту обобщенного типа	Используйте функции предиката типа	18, 19
Описать обобщенный тип без обеспечения реализации	Определите интерфейс с параметром обобщенного типа	22–26

Таблица 12.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
declaration	При включении этого параметра компилятор создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода. Более подробно эти файлы описаны в главе 15
module	Задает формат используемых модулей, как описано в главе 5
outDir	Задает каталог, в который будут помещены файлы JavaScript
rootDir	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

12.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В текущей главе мы продолжим использовать проект `types`, который был создан в главе 7 и дорабатывался до настоящего момента. Чтобы приступить к работе с материалом из этой главы, создайте в папке `src` файл `dataTypes.ts` с содержимым, показанным в листинге 12.1.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 12.1. Содержимое файла `dataTypes.ts` из папки `src`

```
export class Person {
    constructor(public name: string, public city: string) {}
}

export class Product {
    constructor(public name: string, public price: number) {}
}

export class City {
    constructor(public name: string, public population: number) {}
}

export class Employee {
    constructor(public name: string, public role: string) {}
}
```

Замените содержимое файла `index.ts` в папке `src` кодом из листинга 12.2.

Листинг 12.2. Замена содержимого файла `index.ts` из папки `src`

```
import { Person, Product } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

[...people, ...products].forEach(item =>
    console.log('Item: ${item.name}'));
```

В этом листинге используется оператор `import` для объявления зависимостей от классов `Person` и `Product`, определенных в модуле `dataTypes`. Для настройки формата модуля, как описано в главе 5, установите свойство конфигурации `type` в файле `package.json`, как показано в листинге 12.3.

Листинг 12.3. Настройка формата модуля в файле `package.json` из папки `types`

```
{
    "name": "types",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "start": "tsc-watch --onSuccess \"node dist/index.js\""
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
        "tsc-watch": "^6.0.0",
        "typescript": "^5.0.2"
    },
    "type": "module"
}
```

Чтобы компилятор TypeScript использовал файл `package.json` для определения формата модуля и отключил функции, которые больше не нужны, измените свойства конфигурации в соответствии с листингом 12.4.

Листинг 12.4. Настройка компилятора в файле `tsconfig.json` из папки `types`

```
{  
  "compilerOptions": {  
    "target": "ES2022",  
    "outDir": "./dist",  
    "rootDir": "./src",  
    "declaration": true,  
    // "strictNullChecks": true,  
    // "noUncheckedIndexedAccess": true,  
    "module": "Node16"  
  }  
}
```

С помощью консоли выполните в папке `types` команду из листинга 12.5 для запуска компилятора TypeScript, чтобы он автоматически выполнял код после его компиляции.

Листинг 12.5. Запуск компилятора TypeScript

```
npm start
```

Компилятор скомпилирует проект, выполнит вывод, а затем перейдет в режим наблюдения, выдав следующий результат:

```
7:22:32 AM - Starting compilation in watch mode...  
7:22:34 AM - Found 0 errors. Watching for file changes.  
Item: Bob Smith  
Item: Dora Peters  
Item: Running Shoes  
Item: Hat
```

12.2. ПРОБЛЕМЫ, РЕШАЕМЫЕ ОБОБЩЕННЫМИ ТИПАМИ

Лучший способ понять, как работают дженерики и почему они полезны, — это пройти через сценарий, когда обычными типами становится трудно управлять. В листинге 12.6 определен класс, управляющий коллекцией объектов `Person`.

Листинг 12.6. Определение класса в файле `index.ts` из папки `src`

```
import { Person, Product } from "./dataTypes.js";  
  
let people = [new Person("Bob Smith", "London"),  
             new Person("Dora Peters", "New York")];  
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];  
  
class PeopleCollection {  
  private items: Person[] = [];
```

```

constructor(initialItems: Person[]) {
    this.items.push(...initialItems);
}

add(newItem: Person) {
    this.items.push(newItem);
}

getNames(): string[] {
    return this.items.map(item => item.name);
}

getItem(index: number): Person {
    return this.items[index];
}
}

let peopleData = new PeopleCollection(people);

console.log('Names: ${peopleData.getNames().join(", ")}`);
let firstPerson = peopleData.getItem(0);
console.log('First Person: ${firstPerson.name}, ${firstPerson.city}');

```

Класс `PeopleCollection` оперирует объектами `Person`, которые предоставляются через конструктор или метод `add`. Метод `getNames` возвращает массив, содержащий значение `name` каждого объекта `Person`, а метод `getItem` позволяет получить объект `Person` по индексу. Создается новый экземпляр класса `PeopleCollection`, и его методы вызываются для получения следующего вывода:

```

Names: Bob Smith, Dora Peters
First Person: Bob Smith, London

```

12.2.1. Добавление поддержки другого типа

Проблема с классом `PeopleCollection` заключается в том, что он работает только с объектами `Person`. Если мы захотим выполнить тот же набор операций с объектами `Product`, то очевидные решения будут неудобными. Мы могли бы создать новый класс, дублирующий функциональность. Это, конечно, возможно, но в будущем всегда найдется другой тип, с которым придется иметь дело, и вскоре классами станет трудно управлять. Более элегантным решением было бы воспользоваться возможностями TypeScript и адаптировать существующий класс для работы с разными типами, как показано в листинге 12.7.

Листинг 12.7. Добавление поддержки типов в файл index.ts

```

import { Person, Product } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100),
    new Product("Hat", 25)];

type dataType = Person | Product;

```

```

class DataCollection {

    private items: dataType[] = [];

    constructor(initialItems: dataType[]) {
        this.items.push(...initialItems);
    }

    add(newItem: dataType) {
        this.items.push(newItem);
    }

    getNames(): string[] {
        return this.items.map(item => item.name);
    }

    getItem(index: number): dataType {
        return this.items[index];
    }
}

let peopleData = new DataCollection(people);

console.log('Names: ${peopleData.getNames().join(", ")}`);
let firstPerson = peopleData.getItem(0);
if (firstPerson instanceof Person) {
    console.log('First Person: ${firstPerson.name}, ${firstPerson.city}');
}

```

Здесь используется объединение типов для добавления поддержки класса `Product`. Мы могли бы также использовать интерфейс, абстрактный класс или переопределения типов функций, но поддержка более широкого диапазона типов потребовала бы некоторого сужения типов для возврата к конкретному типу. Другая проблема заключается в том, что класс `DataCollection` принимает объекты `Person` и `Product`. В нашем случае нужна поддержка либо объектов `Person`, либо `Product`, но не обоих. Код в листинге 12.7 выдает следующий результат:

```

Names: Bob Smith, Dora Peters
First Person: Bob Smith, London

```

12.3. СОЗДАНИЕ ОБОБЩЕННЫХ КЛАССОВ

Обобщенный класс (generic class) — это класс, обладающий параметризованным типом данных, который представляет собой заполнитель для типа, задаваемого при создании нового объекта с помощью данного класса. Параметры обобщенного типа позволяют писать классы, работающие с определенным типом, не имея заранее информации о том, каким будет этот тип, как показано в листинге 12.8.

Листинг 12.8. Использование обобщенного типа в файле index.ts

```

import { Person, Product } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];

```

```

let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

//type dataType = Person | Product;

class DataCollection<T> {

    private items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    add(newItem: T) {
        this.items.push(newItem);
    }

    // getNames(): string[] {
    //     return this.items.map(item => item.name);
    // }

    getItem(index: number): T {
        return this.items[index];
    }
}

let peopleData = new DataCollection<Person>(people);

//console.log('Names: ${peopleData.getNames().join(", ")}');
let firstPerson = peopleData.getItem(0);
//if (firstPerson instanceof Person) {
//console.log('First Person: ${firstPerson.name}, ${firstPerson.city}');
//}

}

```

Класс `DataCollection` был определен с параметром обобщенного типа, который является частью объявления класса, как показано на рис. 12.1.

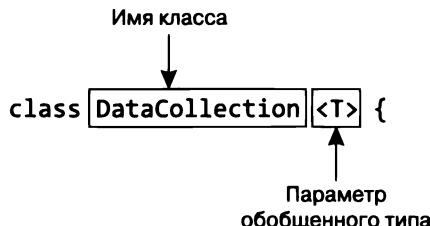


Рис. 12.1. Параметр обобщенного типа

Параметр обобщенного типа задается в угловых скобках символами `<` и `>` (при этом указывается только имя). Общепринято начинать имя параметра типа с буквы `T`, хотя вы вольны использовать любую удобную вам схему именования.

Полученный результат и называется *обобщенным классом*, то есть классом, имеющим по крайней мере один параметр обобщенного типа. В данном примере параметр обобщенного типа обозначен как `T` и может использоваться вместо

конкретного типа. Например, конструктор допускается определить так, чтобы он принимал массив значений T :

```
...
constructor(initialItems: T[]) {
    this.items.push(...initialItems);
}
...
```

Как показывает конструктор, обобщенный тип можно использовать в аннотациях типов, даже если конкретный тип, для которого он является заполнителем, еще неизвестен. Класс в листинге 12.8 определяет единственный параметр типа с именем T и поэтому обозначается как $\text{DataCollection}<T>$, что явно указывает на то, что он является обобщенным классом. Код в листинге 12.8 выдает следующий результат:

```
First Person: Bob Smith, London
```

12.3.1. Аргументы обобщенных типов

При создании экземпляра класса $\text{DataCollection}<T>$ с помощью ключевого слова `new` параметр обобщенного типа преобразуется в конкретный тип с помощью аргумента обобщенного типа, как показано на рис. 12.2.



Рис. 12.2. Создание объекта с использованием аргумента обобщенного типа

В аргументе типа используются угловые скобки, а аргумент в примере задает класс `Person`:

```
...
let peopleData = new DataCollection<Person>(people);
...
```

Этот оператор создает объект $\text{DataCollection}<T>$, где параметром типа T будет `Person`. Когда объект создается на основе обобщенного класса, его тип включает в себя аргумент, например `DataCollection<Person>`. Компилятор TypeScript автоматически применяет `Person` везде, где встречается T . Это означает, что конструктор и метод `add` принимают только объекты `Person`, а вызов метода `getItem` возвращает объект `Person`. TypeScript отслеживает аргумент типа, использованный для создания объекта `DataCollection<Person>`, и никаких утверждений или сужений типа не требуется.

12.3.2. Использование аргументов различных типов

Значение параметра обобщенного типа влияет только на один объект, и при каждом использовании ключевого слова new для аргумента обобщенного типа может использоваться другой тип, в результате чего получается объект `DataCollection<T>`, работающий с другим типом, как показано в листинге 12.9.

Листинг 12.9. Использование аргумента другого типа в файле index.ts из папки src

```
import { Person, Product } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100),
    new Product("Hat", 25)];

class DataCollection<T> {

    private items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    add(newItem: T) {
        this.items.push(newItem);
    }

    // getNames(): string[] {
    //     return this.items.map(item => item.name);
    // }

    getItem(index: number): T {
        return this.items[index];
    }
}

let peopleData = new DataCollection<Person>(people);
let firstPerson = peopleData.getItem(0);
console.log('First Person: ${firstPerson.name}, ${firstPerson.city}');

let productData = new DataCollection<Product>(products);
let firstProduct = productData.getItem(0);
console.log('First Product: ${firstProduct.name}, ${firstProduct.price}'');
```

Новые операторы создают объект `DataCollection<Product>`, используя `Product` в качестве аргумента обобщенного типа. TypeScript отслеживает, какой тип был указан для каждого объекта, и гарантирует, что только этот тип может быть использован. Код в листинге 12.9 выдает следующий результат:

```
First Person: Bob Smith, London
First Product: Running Shoes, 100
```

12.3.3. Ограничение значений обобщенных типов

В листингах 12.8 и 12.9 метод `getNames` закомментирован. По умолчанию в качестве аргумента обобщенного типа может использоваться любой тип, поэтому компилятор рассматривает обобщенные типы как `any`, то есть он не предоставит доступ к свойству `name`, от которого зависит метод `getNames`, без какого-либо сужения типа.

Мы могли бы выполнить сужение диапазона типов в методе `getNames`, но более элегантным подходом является ограничение диапазона типов, которые могут быть использованы в качестве значения параметра обобщенного типа. Это позволяет классу быть инстанцированным только с типами, определяющими функции, на которые опирается обобщенный класс, как показано в листинге 12.10.

Листинг 12.10. Ограничение обобщенных типов в файле index.ts

```
import { Person, Product } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

class DataCollection<T extends (Person | Product)> {
    private items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    add(newItem: T) {
        this.items.push(newItem);
    }

    getNames(): string[] {
        return this.items.map(item => item.name);
    }

    getItem(index: number): T {
        return this.items[index];
    }
}

let peopleData = new DataCollection<Person>(people);
let firstPerson = peopleData.getItem(0);
console.log('First Person: ${firstPerson.name}, ${firstPerson.city}');
console.log('Person Names: ${peopleData.getNames().join(", ")');

let productData = new DataCollection<Product>(products);
let firstProduct = productData.getItem(0);
console.log('First Product: ${firstProduct.name}, ${firstProduct.price}');
console.log('Product Names: ${productData.getNames().join(", ")');
```

Ключевое слово `extends` используется после имени параметра типа для задания ограничения, как показано на рис. 12.3.

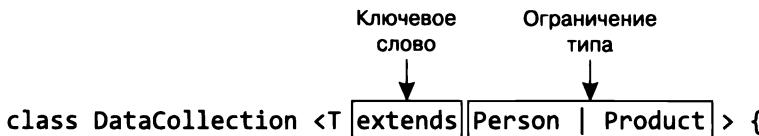


Рис. 12.3. Ограничение параметра обобщенного типа

Изменение в листинге 12.10 можно рассматривать как создание двух уровней ограничений для класса `DataCollection<T>`: первый используется при создании нового объекта, а второй — при использовании объекта.

Первый уровень накладывает ограничения на типы, которые могут быть использованы в качестве аргумента обобщенного типа при создании нового объекта `DataCollection<Product | Person>`. Таким образом, в качестве значения параметра типа могут быть использованы только типы, которые присваиваются `Product | Person`. Этому ограничению удовлетворяют три типа: `Person`, `Product` и объединение `Person | Product`. Они являются единственными допустимыми значениями для параметра обобщенного типа `T`.

Второй уровень ограничений накладывается на значение параметра обобщенного типа при использовании объекта. Например, при создании нового объекта с параметром `Product` в качестве типа `Product` выступает в роли значения `T`: конструктор и добавленные методы принимают только объекты `Product`, а метод `getItem` возвращает только объект `Product`. Если в качестве параметра типа используется `Person`, то `Person` становится значением `T` и типом, используемым конструктором и методами.

Другими словами, ключевое слово `extends` ограничивает типы, которые могут быть присвоены параметру типа, а параметр типа, в свою очередь, ограничивает типы, которые могут быть использованы конкретным экземпляром класса. Поскольку у компилятора есть информация обо всех типах, которые допустимо использовать в качестве параметра обобщенного типа для определения свойства `name`, он позволяет раскомментировать метод `getItem` и прочитать значение свойства `name`, не вызывая ошибок. Код в листинге 12.10 выдает следующий результат:

```

First Person: Bob Smith, London
Person Names: Bob Smith, Dora Peters
First Product: Running Shoes, 100
Product Names: Running Shoes, Hat
  
```

Ограничение обобщенных типов с помощью структуры

Использование объединения типов для ограничения параметров обобщенного типа полезно, но это объединение должно быть расширено для каждого нового типа, который требуется. Альтернативным подходом является использование структуры для ограничения параметра типа, что позволит указать только те свойства, на которые опирается обобщенный класс, как показано в листинге 12.11.

Листинг 12.11. Использование структуры типа в файле index.ts

```

import { City, Person, Product } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100),
    new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];

class DataCollection<T extends { name: string }> {
    private items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    add(newItem: T) {
        this.items.push(newItem);
    }

    getNames(): string[] {
        return this.items.map(item => item.name);
    }

    getItem(index: number): T {
        return this.items[index];
    }
}

let peopleData = new DataCollection<Person>(people);
let firstPerson = peopleData.getItem(0);
console.log('First Person: ${firstPerson.name}, ${firstPerson.city}');
console.log('Person Names: ${peopleData.getNames().join(", ")}`);

let productData = new DataCollection<Product>(products);
let firstProduct = productData.getItem(0);
console.log('First Product: ${firstProduct.name}, ${firstProduct.price}');
console.log('Product Names: ${productData.getNames().join(", ")');

let cityData = new DataCollection<City>(cities);
console.log('City Names: ${cityData.getNames().join(", ")');

```

Структура из листинга 12.11 сообщает компилятору, что экземпляр класса `DataCollection<T>` может быть создан с использованием любого типа, имеющего свойство `name`, которое возвращает строку. Это позволяет создавать объекты `DataCollection` для работы с объектами `Person`, `Product` и `City` без необходимости указывать отдельные типы.

СОВЕТ

Параметры обобщенных типов также могут быть ограничены с помощью псевдонимов типов и интерфейсов. Также можно ограничить обобщенные типы теми, которые определяют конкретную структуру конструктора, что делается с помощью ключевых слов `extends new`, работа с которыми демонстрируется в главе 13.

Код в листинге 12.11 выдает следующий результат:

```
First Person: Bob Smith, London
Person Names: Bob Smith, Dora Peters
First Product: Running Shoes, 100
Product Names: Running Shoes, Hat
City Names: London, Paris
```

12.3.4. Определение нескольких параметров типа

Класс может определять несколько параметров типа. В листинге 12.12 добавляется второй параметр типа в класс `DataCollection<T>`, который используется для корреляции значений данных (также удалены методы класса, которые больше не нужны в этих примерах).

Листинг 12.12. Определение еще одного параметра типа в файле index.ts

```
import { City, Person, Product } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];

class DataCollection<T extends { name: string }, U> {
    private items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    collate(targetData: U[], itemProp: string,
        targetProp: string): (T & U)[] {
        let results = [];
        this.items.forEach(item => {
            let match = targetData.find(d =>
                d[targetProp] === item[itemProp]);
            if (match !== undefined) {
                results.push({ ...match, ...item });
            }
        });
        return results;
    }
}

let peopleData = new DataCollection<Person, City>(people);
let collatedData = peopleData.collate(cities, "city", "name");
collatedData.forEach(c =>
    console.log(`${c.name}, ${c.city}, ${c.population}`));
```

Дополнительные параметры типа разделяются запятыми, как и обычные параметры функции или метода. Класс `DataCollection<T, U>` определяет два параметра обобщенного типа. Новый параметр с именем `U` используется для определения типа аргумента, передаваемого методу `collate`, который сравнивает свойства массива

объектов и пересечения между теми объектами T и U , которые имеют одинаковые значения свойств.

При создании экземпляра обобщенного класса для каждого из параметров обобщенного типа должны быть заданы аргументы, разделенные запятыми, как показано ниже:

```
...
let peopleData = new DataCollection<Person, City>(people);
...
```

Этот оператор создает объект `DataCollection<Person, City>`, который будет хранить объекты `Person` и сравнивать их с объектами `City`. Массив объектов `City` передается в метод `collate`, который сравнивает значения свойства `city` объектов `Person` и свойства `name` объектов `City`.

Свойства объектов, имеющих совпадающие значения, объединяются с помощью синтаксиса оператора `spread` для создания пересечения:

```
...
results.push({ ...match, ...item });
...
```

В нашем примере существует одна пара объектов с совпадающими значениями, и код из листинга 12.12 выдает следующий результат:

```
Bob Smith, London, 8136000
```

Применение параметра типа к методу

Второй параметр типа в листинге 12.12 не так гибок, как хотелось бы, поскольку он требует, чтобы тип данных, используемый методом `collate`, был указан при создании объекта `DataCollection`. То есть это единственный тип данных, который может быть использован с этим методом.

Если тип используется только одним методом, параметр типа может быть вынесен из объявления класса и применен непосредственно к методу, что позволяет при каждом вызове метода указывать разные типы, как показано в листинге 12.13.

Листинг 12.13. Применение параметра типа к методу в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
  new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];
let employees = [new Employee("Bob Smith", "Sales"),
  new Employee("Alice Jones", "Sales")];

class DataCollection<T extends { name: string }> {
  private items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }
}
```

```

collate<U>(targetData: U[], itemProp: string,
            targetProp: string): (T & U)[] {
    let results = [];
    this.items.forEach(item => {
        let match = targetData.find(d =>
            d[targetProp] === item[itemProp]);
        if (match !== undefined) {
            results.push({ ...match, ...item });
        }
    });
    return results;
}

let peopleData = new DataCollection<Person>(people);
let collatedData = peopleData.collate<City>(cities, "city", "name");
collatedData.forEach(c =>
    console.log(`${c.name}, ${c.city}, ${c.population}`));
let empData = peopleData.collate<Employee>(employees, "name", "name");
empData.forEach(c => console.log(`${c.name}, ${c.city}, ${c.role}`));

```

Параметр типа `U` применяется непосредственно к методу `collate`, предоставляя возможность указывать тип при вызове метода, например, так:

```

...
let collatedData = peopleData.collate<City>(cities, "city", "name");
...

```

Параметр типа метода позволяет вызывать метод `collate` с использованием объектов `City`, а затем снова вызывать его с объектами `Employee`. Код, приведенный в листинге 12.13, выдает следующий результат:

```

Bob Smith, London, 8136000
Bob Smith, London, Sales

```

12.3.5. Разрешение компилятору выводить аргументы типа

Компилятор TypeScript способен выводить аргументы обобщенного типа на основе способа создания объектов или вызова методов. Это удобный способ написания лаконичного кода. Однако сначала необходимо убедиться, что объекты инициализируются теми типами, которые были бы явно указаны. В листинге 12.14 создается экземпляр класса `DataCollection<T>` и вызывается метод `collate` без явного указания аргументов типа, оставляя компилятору возможность самостоятельно вывести тип.

Листинг 12.14. Вывод обобщенного типа в файле index.ts

```

import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
              new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];

```

```

let employees = [new Employee("Bob Smith", "Sales"),
  new Employee("Alice Jones", "Sales")];

class DataCollection<T extends { name: string }> {
  private items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  collate<U>(targetData: U[], itemProp: string,
    targetProp: string): (T & U)[] {
    let results = [];
    this.items.forEach(item => {
      let match = targetData.find(d =>
        d[targetProp] === item[itemProp]);
      if (match !== undefined) {
        results.push({ ...match, ...item });
      }
    });
    return results;
  }
}

export let peopleData = new DataCollection(people);
export let collatedData = peopleData.collate(cities, "city", "name");
collatedData.forEach(c =>
  console.log(`${c.name}, ${c.city}, ${c.population}`));
export let empData = peopleData.collate(employees, "name", "name");
empData.forEach(c => console.log(`${c.name}, ${c.city}, ${c.role}`));

```

Компилятор способен вывести аргументы типа на основе аргумента, передаваемого конструктору `DataCollection<T>`, и первого аргумента, передаваемого методу `collate`. Для проверки типов, выведенных компилятором, необходимо просмотреть файл `index.d.ts` в папке `dist`, который генерируется при включении опции `declaration`.

СОВЕТ

В проектах, использующих модули, файлы, созданные с помощью опции `declaration`, содержат только те типы, которые экспортируются за пределами модуля, поэтому в листинге 12.14 я добавил ключевое слово `export`.

Вот типы, выведенные компилятором:

```

...
export declare let peopleData: DataCollection<Person>;
export declare let collatedData: (Person & City)[];
export declare let empData: (Person & Employee)[];
...

```

Код в листинге 12.14 выдает следующий результат:

```

Bob Smith, London, 8136000
Bob Smith, London, Sales

```

12.3.6. Расширение обобщенных классов

Обобщенный класс может быть расширен, и для подкласса допускается выбрать несколько способов работы с параметрами обобщенного типа, как описано в следующих разделах.

Добавление дополнительных функций к существующим параметрам типа

Первый подход заключается в том, чтобы просто добавить новые функции к тем, которые уже определены родительским классом (суперклассом), используя те же обобщенные типы, как показано в листинге 12.15.

Листинг 12.15. Создание подкласса обобщенного класса в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];
let employees = [new Employee("Bob Smith", "Sales"),
    new Employee("Alice Jones", "Sales")];

class DataCollection<T extends { name: string }> {
    protected items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    collate<U>(targetData: U[], itemProp: string, targetProp: string): (T & U)[] {
        let results = [];
        this.items.forEach(item => {
            let match = targetData.find(d =>
                d[targetProp] === item[itemProp]);
            if (match !== undefined) {
                results.push({ ...match, ...item });
            }
        });
        return results;
    }
}

class SearchableCollection<T extends { name: string }>
    extends DataCollection<T> {

    constructor(initialItems: T[]) {
        super(initialItems);
    }

    find(name: string): T | undefined {
        return this.items.find(item => item.name === name);
    }
}
```

```
let peopleData = new SearchableCollection<Person>(people);
let foundPerson = peopleData.find("Bob Smith");
if (foundPerson !== undefined) {
  console.log('Person ${ foundPerson.name }, ${ foundPerson.city}');
}
```

Класс `SearchableCollection<T>` является производным от `DataCollection<T>` и определяет метод `find`, позволяющий находить объект по его свойству `name`. В объявлении класса `SearchableCollection<T>` используется ключевое слово `extends`, а также включаются параметры типа, например, так:

```
...
class SearchableCollection<T extends { name: string }>
  extends DataCollection<T> {
...
}
```

Тип обобщенного класса включает в себя его параметры типа, так что суперкласс является `DataCollection<T>`. Параметр типа, определяемый классом `SearchableCollection<T>`, должен быть совместим с параметром типа родительского класса, поэтому мы используем ту же структуру типа для указания типов, определяющих свойство `name`.

СОВЕТ

Обратите внимание, что в листинге 12.15 изменилось ключевое слово контроля доступа в свойстве `items` на `protected`, что позволило подклассам обращаться к нему. Подробнее о ключевых словах управления доступом в TypeScript — в главе 11.

Класс `SearchableCollection<T>` создается так же, как и любой другой, с использованием аргумента типа (или с возможностью компилятора вывести этот аргумент). Код в листинге 12.15 выдает следующий результат:

```
Person Bob Smith, London
```

Исправление параметра обобщенного типа

Для некоторых классов необходимо определить функциональность, доступную только при использовании подмножества типов, поддерживаемых суперклассом. В подобных ситуациях подкласс может использовать фиксированный тип для параметра типа родительского класса. В этом случае подкласс не будет обобщенным классом (листинг 12.16).

Листинг 12.16. Исправление параметра обобщенного типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
  new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100),
  new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];
let employees = [new Employee("Bob Smith", "Sales"),
  new Employee("Alice Jones", "Sales")];
```

```

class DataCollection<T extends { name: string }> {
    protected items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    collate<U>(targetData: U[], itemProp: string,
        targetProp: string): (T & U)[] {
        let results = [];
        this.items.forEach(item => {
            let match = targetData.find(d =>
                d[targetProp] === item[itemProp]);
            if (match !== undefined) {
                results.push({ ...match, ...item });
            }
        });
        return results;
    }
}

class SearchableCollection extends DataCollection<Employee> {

    constructor(initialItems: Employee[]) {
        super(initialItems);
    }

    find(searchTerm: string): Employee[] {
        return this.items.filter(item =>
            item.name === searchTerm || item.role === searchTerm);
    }
}

let employeeData = new SearchableCollection(employees);
employeeData.find("Sales").forEach(e =>
    console.log(`Employee ${e.name}, ${e.role}`));

```

Класс `SearchableCollection` расширяет `DataCollection<Employee>`, где исправлен параметр обобщенного типа, позволяя `SearchableCollection` работать только с объектами класса `Employee`. Для создания объекта `SearchableCollection` нельзя использовать параметр типа, и код в методе `find` может безопасно обращаться к свойствам, которые определены в классе `Employee`. Код в листинге 12.16 выдает следующий результат:

```

Employee Bob Smith, Sales
Employee Alice Jones, Sales

```

Ограничение параметра обобщенного типа

Третий подход помогает найти компромисс между двумя предыдущими примерами, предоставляя переменную обобщенного типа, но ограничивая ее конкретными типами, как показано в листинге 12.17. Это позволяет реализовать функциональность, которая может зависеть от особенностей конкретных классов, не фиксируя полностью параметр типа.

Листинг 12.17. Ограничение параметра типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100),
    new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];
let employees = [new Employee("Bob Smith", "Sales"),
    new Employee("Alice Jones", "Sales")];

class DataCollection<T extends { name: string }> {
    protected items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    collate<U>(targetData: U[], itemProp: string,
        targetProp: string): (T & U)[] {
        let results = [];
        this.items.forEach(item => {
            let match = targetData.find(d =>
                d[targetProp] === item[itemProp]);
            if (match !== undefined) {
                results.push({ ...match, ...item });
            }
        });
        return results;
    }
}

class SearchableCollection<T
    extends Employee | Person> extends DataCollection<T> {

    constructor(initialItems: T[]) {
        super(initialItems);
    }

    find(searchTerm: string): T[] {
        return this.items.filter(item => {
            if (item instanceof Employee) {
                return item.name ===
                    searchTerm || item.role === searchTerm;
            } else if (item instanceof Person) {
                return item.name ===
                    searchTerm || item.city === searchTerm;
            }
        });
    }
}

let employeeData = new SearchableCollection<Employee>(employees);
employeeData.find("Sales").forEach(e =>
    console.log('Employee ${ e.name }, ${ e.role}'));
```

Параметр типа, задаваемый подклассом, должен быть совместим с параметром типа, который он наследует, то есть может быть использован только более ограничительный тип. В нашем примере объединение `Employee | Person` может быть присвоено структуре, используемой для ограничения параметра типа `DataCollection<T>`.

ВНИМАНИЕ

Следует помнить, что если объединение используется для ограничения параметра обобщенного типа, то это объединение само по себе является допустимым аргументом для этого параметра. То есть класс `SearchableCollection` в листинге 12.17 может быть создан с параметром типа `Employee`, `Product` и `Employee | Product`. Подробнее о расширенных возможностях ограничения аргументов типа – в главе 13.

Метод `find` использует ключевое слово `instanceof` для сужения объектов до определенных типов, чтобы провести сравнение значений свойств. Код в листинге 12.17 выдает следующий результат:

```
Employee Bob Smith, Sales
Employee Alice Jones, Sales
```

12.3.7. Защита обобщенных типов

Класс `SearchableCollection<T>` в листинге 12.17 использовал ключевое слово `instanceof` для идентификации объектов `Employee` и `Person`. Это легко управляемо, поскольку ограничение, наложенное на параметр типа, означает, что существует лишь небольшое количество типов, с которыми приходится взаимодействовать. Для классов с параметром типа, не имеющим ограничений, сужение диапазона до конкретного типа может быть затруднено (листинг 12.18).

Листинг 12.18. Сужение обобщенного типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];
let employees = [new Employee("Bob Smith", "Sales"),
    new Employee("Alice Jones", "Sales")];

class DataCollection<T> {
    protected items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    filter<V extends T>(): V[] {
        return this.items.filter(item => item instanceof V) as V[];
    }
}
```

```

let mixedData
  = new DataCollection<Person | Product>([...people, ...products]);
let filteredProducts = mixedData.filter<Product>();
filteredProducts.forEach(p =>
  console.log('Product: ${ p.name }, ${p.price}'));

```

В листинге 12.18 представлен метод `filter`, использующий ключевое слово `instanceof` для выбора объектов определенного типа из массива элементов данных. Создается объект `DataCollection<Person | Product>` с массивом, содержащим набор объектов `Person` и `Product`. Затем новый метод `filter` используется для выбора объектов `Product`.

СОВЕТ

Обратите внимание, что параметр обобщенного типа метода `filter`, обозначенный как `V`, ограничен ключевым словом `extend`, то есть компилятор может принимать только типы, которые могут быть присвоены классу обобщенного типа `T`. Это не позволяет компилятору рассматривать `V` как `any`.

Этот пример не компилируется и выдает следующее сообщение об ошибке:

```
src/index.ts(18,58): error TS2693: 'V' only refers to a type, but is being
used as a value here.
```

Ни одна функция JavaScript не эквивалентна обобщенным типам, поэтому они удаляются из кода TypeScript в процессе компиляции. Это означает, что во время выполнения нет информации для использования обобщенных типов с ключевым словом `instanceof`.

В ситуациях, когда необходимо идентифицировать объекты по типу, обобщенные типы бесполезны и придется использовать функцию предиката. В листинге 12.19 в метод `filter` добавляется параметр, принимающий функцию предиката типа, которая затем используется для поиска объектов определенного типа.

Листинг 12.19. Использование функции предиката типа в файле index.ts

```

import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
  new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];
let employees = [new Employee("Bob Smith", "Sales"),
  new Employee("Alice Jones", "Sales")];

class DataCollection<T> {
  protected items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  filter<V extends T>(predicate: (target) => target is V): V[] {
    return this.items.filter(item => predicate(item)) as V[];
  }
}

```

```

let mixedData
  = new DataCollection<Person | Product >([...people, ...products]);
function isProduct(target): target is Product {
  return target instanceof Product;
}
let filteredProducts = mixedData.filter<Product>(isProduct);
filteredProducts.forEach(p =>
  console.log('Product: ${ p.name}, ${p.price}'));

```

Функция предиката для требуемого типа передается в качестве аргумента методу `filter`, используя возможности JavaScript, доступные при выполнении кода. Это предоставляет методу средства для отбора требуемых объектов. Код в листинге 12.19 дает следующий результат:

```

Product: Running Shoes, 100
Product: Hat, 25

```

12.3.8. Определение статического метода в обобщенном классе

Только свойства и методы экземпляра имеют обобщенный тип, который может различаться для каждого объекта. Доступ к статическим методам осуществляется через класс, как показано в листинге 12.20.

Листинг 12.20. Определение статического метода в файле index.ts

```

import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
  new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];
let employees = [new Employee("Bob Smith", "Sales"),
  new Employee("Alice Jones", "Sales")];

class DataCollection<T> {
  protected items: T[] = [];

  constructor(initialItems: T[]) {
    this.items.push(...initialItems);
  }

  filter<V extends T>(predicate: (target) => target is V): V[] {
    return this.items.filter(item => predicate(item)) as V[];
  }

  static reverse(items: any[]): {
    return items.reverse();
  }
}

let mixedData
  = new DataCollection<Person | Product >([...people, ...products]);

```

```

function isProduct(target): target is Product {
    return target instanceof Product;
}

let filteredProducts = mixedData.filter<Product>(isProduct);
filteredProducts.forEach(p =>
    console.log('Product: ${ p.name}, ${p.price}'));

let reversedCities: City[] = DataCollection.reverse(cities);
reversedCities.forEach(c =>
    console.log('City: ${c.name}, ${c.population}'));

```

Доступ к обратному методу `static` осуществляется через класс `DataCollection` без использования аргумента типа, например, так:

```

...
let reversedCities: City[] = DataCollection.reverse(cities);
...

```

Статические методы могут определять свои собственные параметры обобщенного типа, как показано в листинге 12.21.

Листинг 12.21. Добавление параметра типа в файл index.ts

```

import { City, Person, Product, Employee } from "./dataTypes.js";

let people = [new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York")];
let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];
let cities = [new City("London", 8136000), new City("Paris", 2141000)];
let employees = [new Employee("Bob Smith", "Sales"),
    new Employee("Alice Jones", "Sales")];

class DataCollection<T> {
    protected items: T[] = [];

    constructor(initialItems: T[]) {
        this.items.push(...initialItems);
    }

    filter<V extends T>(predicate: (target) => target is V): V[] {
        return this.items.filter(item => predicate(item)) as V[];
    }

    static reverse<ArrayType>(items: ArrayType[]): ArrayType[] {
        return items.reverse();
    }
}

let mixedData
= new DataCollection<Person | Product >([...people, ...products]);

function isProduct(target): target is Product {
    return target instanceof Product;
}

```

```
let filteredProducts = mixedData.filter<Product>(isProduct);
filteredProducts.forEach(p =>
  console.log('Product: ${ p.name}, ${p.price}'));
  
let reversedCities = DataCollection.reverse<City>(cities);
reversedCities.forEach(c =>
  console.log('City: ${c.name}, ${c.population}'));
```

Метод `reverse` определяет параметр типа, задающий тип массива, который он обрабатывает. При вызове метода через класс `DataCollection` после имени метода указывается аргумент типа, как в данном случае:

```
...
let reversedCities = DataCollection.reverse<City>(cities);
...
```

Параметры типа, определяемые статическими методами, отличаются от параметров, определяемых классом для использования в свойствах и методах экземпляра. Код в листинге 12.21 выдает следующий результат:

```
Product: Running Shoes, 100
Product: Hat, 25
City: Paris, 2141000
City: London, 8136000
```

12.4. ОПРЕДЕЛЕНИЕ ОБОБЩЕННЫХ ИНТЕРФЕЙСОВ

Интерфейсы могут быть определены с параметрами обобщенного типа, что позволяет определить функциональность без указания отдельных типов. В листинге 12.22 определен интерфейс с параметром обобщенного типа.

Листинг 12.22. Определение обобщенного интерфейса в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type shapeType = { name: string };

interface Collection<T extends shapeType> {

  add(...newItems: T[]): void;
  get(name: string): T;
  count: number;
}
```

Интерфейс `Collection<T>` имеет параметр обобщенного типа с именем `T`, что соответствует синтаксису, используемому для параметров типа в классе. Параметр типа используется методами `add` и `get`, при этом на него накладывается ограничение, требующее, чтобы принимаемые типы имели свойство `name`.

Интерфейс с параметром обобщенного типа описывает набор абстрактных операций, но не указывает, над какими типами они могут совершаться, оставляя выбор конкретных типов за производными интерфейсами или классами реализации. Код в листинге 12.22 не дает никакого результата.

12.4.1. Расширение обобщенных интерфейсов

Обобщенные интерфейсы могут быть расширены так же, как и обычные интерфейсы, и возможности работы с параметрами их типов такие же, как и при расширении общих классов. В листинге 12.23 показан набор интерфейсов, расширяющих интерфейс `Collection<T>`.

Листинг 12.23. Расширение общего интерфейса в файле `index.ts`

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type shapeType = { name: string };

interface Collection<T extends shapeType> {

    add(...newItems: T[]): void;
    get(name: string): T;
    count: number;
}

interface SearchableCollection<T extends shapeType> extends Collection<T> {

    find(name: string): T | undefined;
}

interface ProductCollection extends Collection<Product> {

    sumPrices(): number;
}

interface PeopleCollection<T extends Product | Employee>
    extends Collection<T> {

    getNames(): string[];
}
```

Код в листинге 12.23 не выдает никаких результатов.

12.4.2. Реализация обобщенного интерфейса

Когда класс реализует обобщенный интерфейс, он должен реализовать все свойства и методы интерфейса. Однако у него есть выбор, как обрабатывать параметры типа, о чем говорится в следующих разделах. Некоторые из этих вариантов аналогичны тем, которые используются при расширении обобщенных классов и интерфейсов.

Передача параметра обобщенного типа

Самый простой подход – реализовать свойства и методы интерфейса, не изменяя параметра типа. Это достигается путем создания обобщенного класса, который непосредственно реализует интерфейс, как показано в листинге 12.24.

Листинг 12.24. Реализация интерфейса в файле index.ts

```

import { City, Person, Product, Employee } from "./dataTypes.js";

type shapeType = { name: string };

interface Collection<T extends shapeType> {

    add(...newItems: T[]): void;
    get(name: string): T;
    count: number;
}

class ArrayCollection<DataType extends shapeType>
    implements Collection<DataType> {
    private items: DataType[] = [];

    add(...newItems): void {
        this.items.push(...newItems);
    }

    get(name: string): DataType {
        return this.items.find(item => item.name === name);
    }

    get count(): number {
        return this.items.length;
    }
}

let peopleCollection: Collection<Person> = new ArrayCollection<Person>();
peopleCollection.add(new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York"));
console.log('Collection size: ${peopleCollection.count}');

```

Класс `ArrayCollection<DataType>` использует ключевое слово `implements`, чтобы объявить, что он соответствует интерфейсу. Так как интерфейс имеет параметр обобщенного типа, класс `ArrayCollection<DataType>` должен определить совместимый параметр. Поскольку параметр типа для интерфейса должен иметь свойство `name`, то и параметру типа для класса необходимо иметь это свойство. Для обеспечения согласованности в нашем случае был использован один и тот же псевдоним типа для интерфейса и класса.

Класс `ArrayCollection<DataType>` требует аргумент типа при создании объекта и может работать с ним через интерфейс `Collection<T>`, например, так:

```

...
let peopleCollection: Collection<Person> = new ArrayCollection<Person>();
...

```

Аргумент типа позволяет установить связь между обобщенным типом для класса и реализуемым им интерфейсом, так что объект `ArrayCollection<Person>` реализует интерфейс `Collection<Person>`. Код в листинге 12.24 выдает следующий результат:

```
Collection size: 2
```

Ограничение или исправление параметра обобщенного типа

Классы могут предоставлять реализацию интерфейса, специфичную для определенного типа или подмножества типов, поддерживаемых этим интерфейсом (листинг 12.25).

Листинг 12.25. Реализация интерфейса в файле index.ts из папки src

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type shapeType = { name: string };

interface Collection<T extends shapeType> {

    add(...newItems: T[]): void;
    get(name: string): T;
    count: number;
}

class PersonCollection implements Collection<Person> {
    private items: Person[] = [];

    add(...newItems: Person[]): void {
        this.items.push(...newItems);
    }

    get(name: string): Person {
        return this.items.find(item => item.name === name);
    }

    get count(): number {
        return this.items.length;
    }
}

let peopleCollection: Collection<Person> = new PersonCollection();
peopleCollection.add(new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York"));
console.log('Collection size: ${peopleCollection.count}');
```

Класс `PersonCollection` реализует интерфейс `Collection<Product>`, и код в листинге 12.25 при компиляции и выполнении выдает следующий результат:

```
Collection size: 2
```

Создание реализации абстрактного интерфейса

Абстрактный класс может предоставлять частичную реализацию интерфейса, которая может быть дополнена подклассами. Абстрактный класс имеет те же возможности работы с параметрами типа, что и обычные классы: передавать их подклассам без изменений, накладывать дополнительные ограничения или исправлять конкретные типы. В листинге 12.26 показан абстрактный класс, передающий аргумент обобщенного типа интерфейса.

Листинг 12.26. Определение абстрактного класса в файле index.ts

```

import { City, Person, Product, Employee } from "./dataTypes.js";

type shapeType = { name: string };

interface Collection<T extends shapeType> {

    add(...newItems: T[]): void;
    get(name: string): T;
    count: number;
}

abstract class ArrayCollection<T extends shapeType>
    implements Collection<T> {
    protected items: T[] = [];

    add(...newItems: T[]): void {
        this.items.push(...newItems);
    }

    abstract get(searchTerm: string): T;

    get count(): number {
        return this.items.length;
    }
}

class ProductCollection extends ArrayCollection<Product> {

    get(searchTerm: string): Product {
        return this.items.find(item => item.name === searchTerm);
    }
}

class PersonCollection extends ArrayCollection<Person> {

    get(searchTerm: string): Person {
        return this.items.find(item =>
            item.name === searchTerm || item.city === searchTerm);
    }
}

let peopleCollection: Collection<Person> = new PersonCollection();
peopleCollection.add(new Person("Bob Smith", "London"),
    new Person("Dora Peters", "New York"));
let productCollection: Collection<Product> = new ProductCollection();
productCollection.add(new Product("Running Shoes", 100),
    new Product("Hat", 25));
[peopleCollection, productCollection].forEach(c =>
    console.log('Size: ${c.count}'));

```

Класс `ArrayCollection<T>` является абстрактным и предоставляет частичную реализацию интерфейса `Collection<T>`, оставляя подклассам возможность реализовать метод `get`. Классы `ProductCollection` и `PersonCollection` расширяют

`ArrayList<T>`, сужая параметр обобщенного типа до конкретных типов и реализуя метод `get` для работы со свойствами соответствующего типа. Код в листинге 12.26 выдает следующий результат:

```
Size: 2
Size: 2
```

РЕЗЮМЕ

В этой главе вы познакомились с обобщенными типами, мы рассмотрели проблему, которую они решают. Была продемонстрирована связь между параметрами и аргументами обобщенных типов, а также различные способы ограничения или исправления обобщенных типов. Вы узнали, что обобщенные типы могут использоваться с обычными и абстрактными классами и интерфейсами, а также как функции и методы могут иметь обобщенные типы, которые разрешаются при каждом их использовании.

- Параметры обобщенного типа — это заполнители для типов, которые указываются при создании класса с помощью аргумента типа.
- Параметры обобщенного типа могут быть ограничены таким образом, что классы могут создаваться только с аргументами типа, соответствующими этим ограничениям.
- Обобщенные типы можно защитить с помощью функций предиката.
- Обобщенные типы могут использоваться с интерфейсами и наследуются классами реализации.

В следующей главе мы подробнее поговорим о возможностях обобщенных типов, которые предоставляет TypeScript.

13

Расширенные возможности обобщенных типов

В этой главе

- ✓ Использование типизированных коллекций JavaScript с параметрами обобщенных типов.
- ✓ Итерация по типобезопасной коллекции.
- ✓ Создание ключей коллекции с индексными типами.
- ✓ Преобразование типов с помощью сопоставлений.
- ✓ Использование встроенных сопоставлений типов.
- ✓ Выбор обобщенных типов с помощью условных выражений типа.

В этой главе продолжается описание возможностей обобщенных типов, предоставляемых TypeScript, и уделено особое внимание расширенным функциям. Здесь вы найдете информацию о том, как использовать обобщенные типы с коллекциями и итераторами, познакомитесь с индексными типами и функциями сопоставления типов. Также будет представлена наиболее гибкая из возможностей обобщенных типов — условные типы. В табл. 13.1 приведено краткое содержание главы.

В табл. 13.2 перечислены параметры компилятора TypeScript, используемые в данной главе.

Таблица 13.1. Краткое содержание главы

Задача	Решение	Листинг
Использовать классы коллекций типобезопасным образом	Укажите аргумент обобщенного типа при создании коллекции	3, 4
Использовать итераторы типобезопасным образом	Используйте интерфейсы, поддерживающие аргументы обобщенного типа	5–7
Определить тип, значением которого может быть только имя свойства	Используйте запросы индексного типа	8–14
Преобразовать тип	Используйте сопоставление типов	15–22
Выбрать тип программным способом	Используйте условные типы	23–32

Таблица 13.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
declaration	При включении этого параметра компилятор создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода. Более подробно эти файлы описаны в главе 15
downlevelIteration	Включает поддержку итераторов при работе со старыми версиями JavaScript
outDir	Задает каталог, в который будут помещены файлы JavaScript
rootDir	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

13.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В этой главе мы продолжим использовать проект `types`, созданный ранее. Чтобы приступить к работе с материалом из данной главы, замените содержимое файла `index.ts` в папке `src` кодом из листинга 13.1.

Листинг 13.1. Замена содержимого файла `index.ts` из папки `src`

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {
```

```

constructor(private items: T[] = []) {}
add(...newItems: T[]): void {
    this.items.push(...newItems);
}
get(name: string): T {
    return this.items.find(item => item.name === name);
}
get count(): number {
    return this.items.length;
}
}

let productCollection: Collection<Product> = new Collection(products);
console.log('There are ${ productCollection.count } products');
let p = productCollection.get("Hat");
console.log('Product: ${ p.name }, ${ p.price }');

```

Откройте новое окно командной строки, перейдите в папку `types` и выполните команду, показанную в листинге 13.2, для запуска компилятора TypeScript, чтобы он автоматически выполнял код после его компиляции.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 13.2. Запуск компилятора TypeScript

```
npm start
```

Компилятор скомпилирует проект, выполнит вывод, а затем перейдет в режим наблюдения, выдав следующий результат:

```

7:31:10 AM - Starting compilation in watch mode...
7:31:11 AM - Found 0 errors. Watching for file changes.
There are 2 products
Product: Hat, 25

```

13.2. ИСПОЛЬЗОВАНИЕ ОБОБЩЕННЫХ КОЛЛЕКЦИЙ

В TypeScript предусмотрена поддержка использования коллекций JavaScript с параметрами обобщенных типов, что позволяет обобщенному классу безопасно оперировать коллекциями (табл. 13.3). Классы коллекций JavaScript описаны в главе 4.

Таблица 13.3. Типы обобщенных коллекций

Название	Описание
Map<K, V>	Структура данных Map, где K — тип ключа, V — тип значения
ReadonlyMap<K, V>	Структура данных Map, которая не может быть изменена
Set<T>	Множество Set, где T — тип значения
ReadonlySet<T>	Множество Set, которое не может быть изменено

В листинге 13.3 показано, как обобщенный класс может использовать параметры своего типа с коллекцией.

Листинг 13.3. Использование коллекции в файле index.ts из папки src

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {
    private items: Set<T>;

    constructor(initialItems: T[] = []) {
        this.items = new Set<T>(initialItems);
    }

    add(...newItems: T[]): void {
        newItems.forEach(newItem => this.items.add(newItem));
    }

    get(name: string): T {
        return [...this.items.values()].find(item => item.name === name);
    }

    get count(): number {
        return this.items.size;
    }
}

let productCollection: Collection<Product> = new Collection(products);
console.log('There are ${ productCollection.count } products');
let p = productCollection.get("Hat");
console.log('Product: ${ p.name }, ${ p.price }');
```

Для хранения элементов мы заменили класс `Collection<T>` на `Set<T>`, используя параметр обобщенного типа для этой коллекции. Компилятор TypeScript использует параметр типа для предотвращения добавления в набор других типов данных, и при извлечении объектов из коллекции не требуется защита типа. Аналогичный подход можно применить и к карте (мап), как показано в листинге 13.4.

Листинг 13.4. Структура данных мап в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {
    private items: Map<string, T>;

    constructor(initialItems: T[] = []) {
        this.items = new Map<string, T>();
        this.add(...initialItems);
    }
}
```

```

add(...newItems: T[]): void {
    newItems.forEach(newItem => this.items.set(newItem.name, newItem));
}

get(name: string): T {
    return this.items.get(name);
}

get count(): number {
    return this.items.size;
}
}

let productCollection: Collection<Product> = new Collection(products);
console.log('There are ${ productCollection.count } products');
let p = productCollection.get("Hat");
console.log('Product: ${ p.name }, ${ p.price }');

```

Обобщенные классы не обязаны предоставлять параметры обобщенного типа для коллекций. Вместо этого можно указывать конкретные типы. В приведенном примере для хранения объектов используется `Map`, где свойство `name` выступает в качестве ключа. Безопасность применения свойства `name` обеспечивается ограничением, наложенным на параметр типа с именем `T`. Код в листинге 13.4 выдает следующий результат:

```

There are 2 products
Product: Hat, 25

```

13.3. ИСПОЛЬЗОВАНИЕ ОБОБЩЕННЫХ ИТЕРАТОРОВ

Как объяснялось в главе 4, итераторы позволяют перечислять последовательность значений. Поддержка итераторов является общей функцией для классов, работающих с другими типами, например коллекциями. В табл. 13.4 перечислены интерфейсы TypeScript, предназначенные для описания итераторов и их результатов.

Таблица 13.4. Интерфейс итератора TypeScript

Название	Описание
<code>Iterator<T></code>	Описывает итератор, метод <code>next</code> которого возвращает объекты <code>IteratorResult<T></code>
<code>IteratorResult<T></code>	Описывает результат, получаемый итератором, со свойствами <code>done</code> и <code>value</code>
<code>Iterable<T></code>	Определяет объект, имеющий свойство <code>Symbol.iterator</code> , которое поддерживает итерацию напрямую
<code>IterableIterator<T></code>	Объединяет интерфейсы <code>Iterator<T></code> и <code>Iterable<T></code> для описания объекта, имеющего свойство <code>Symbol.iterator</code> и определяющего метод <code>next</code> и свойство <code>result</code>

В листинге 13.5 показано применение интерфейсов `Iterator<T>` и `IteratorResult<T>` для предоставления доступа к содержимому `Map<string, T>`, используемого для хранения объектов класса `Collection<T>`.

Листинг 13.5. Итерация объектов в файле `index.ts` из папки `src`

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {
    private items: Map<string, T>;

    constructor(initialItems: T[] = []) {
        this.items = new Map<string, T>();
        this.add(...initialItems);
    }

    add(...newItems: T[]): void {
        newItems.forEach(newItem => this.items.set(newItem.name, newItem));
    }

    get(name: string): T {
        return this.items.get(name);
    }

    get count(): number {
        return this.items.size;
    }

    values(): Iterator<T> {
        return this.items.values();
    }
}

let productCollection: Collection<Product> = new Collection(products);
console.log('There are ${ productCollection.count } products');

let iterator: Iterator<Product> = productCollection.values();
let result: IteratorResult<Product> = iterator.next();
while (!result.done) {
    console.log(`Product: ${result.value.name}, ${ result.value.price}`);
    result = iterator.next();
}
```

Метод `values`, определенный классом `Collection<T>`, возвращает итератор `Iterator<T>`. Когда этот метод вызывается для объекта `Collection<Product>`, возвращаемый им итератор будет создавать объекты `IteratorResult<Product>` с помощью своего метода `next`. Свойство `result` каждого объекта `IteratorResult<Product>` будет возвращать `Product`, что позволит выполнять итерацию объектов, управляемых коллекцией. Код в листинге 13.5 выдает следующий результат:

```
There are 2 products
Product: Running Shoes, 100
Product: Hat, 25
```

ИСПОЛЬЗОВАНИЕ ИТЕРАТОРОВ В JAVASCRIPT ES5 И БОЛЕЕ РАННИХ ВЕРСИЯХ

Итераторы были введены в стандарте JavaScript ES6. Если вы используете итераторы в своем проекте и ориентируетесь на более ранние версии JavaScript, то необходимо установить свойство компилятора TypeScript `downlevelIteration` в значение `true`.

13.3.1. Объединение итерируемого и итератора

Интерфейс `IterableIterator<T>` допускается использовать для описания объектов, которые могут быть итерированы и которые также определяют свойство `Symbol.iterator`. Объекты, реализующие этот интерфейс, можно перечислить более элегантно (листинг 13.6).

Листинг 13.6. Использование итератора в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> {

    private items: Map<string, T>;

    constructor(initialItems: T[] = []) {
        this.items = new Map<string, T>();
        this.add(...initialItems);
    }

    add(...newItems: T[]): void {
        newItems.forEach(newItem => this.items.set(newItem.name, newItem));
    }

    get(name: string): T {
        return this.items.get(name);
    }

    get count(): number {
        return this.items.size;
    }

    values(): IterableIterator<T> {
        return this.items.values();
    }
}

let productCollection: Collection<Product> = new Collection(products);
console.log('There are ${ productCollection.count } products');
[...productCollection.values()].forEach(p =>
    console.log('Product: ${p.name}, ${ p.price}'));

```

Метод `values` возвращает объект `IterableIterator`. Это возможно благодаря тому, что результат метода `Map` определяет все члены, заданные интерфейсом. Комбинированный интерфейс позволяет напрямую итерировать результат метода `values`. В приведенном примере с помощью оператора `spread` заполняется массив, а затем перечисляется его содержимое с помощью метода `forEach`. Код в листинге 13.6 выдает следующий результат:

```
There are 2 products
Product: Running Shoes, 100
Product: Hat, 25
```

13.3.2. Создание итерируемого класса

Классы, определяющие свойство `Symbol.iterator`, могут реализовать интерфейс `Iterable<T>`, который позволяет выполнять итерацию без необходимости вызова метода или чтения свойства, как показано в листинге 13.7.

Листинг 13.7. Создание итерируемого класса в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

type shapeType = { name: string };

class Collection<T extends shapeType> implements Iterable<T> {
    private items: Map<string, T>;

    constructor(initialItems: T[] = []) {
        this.items = new Map<string, T>();
        this.add(...initialItems);
    }

    add(...newItems: T[]): void {
        newItems.forEach newItem => this.items.set(newItem.name, newItem);
    }

    get(name: string): T {
        return this.items.get(name);
    }

    get count(): number {
        return this.items.size;
    }

    [Symbol.iterator](): Iterator<T> {
        return this.items.values();
    }
}

let productCollection: Collection<Product> = new Collection(products);
console.log('There are ${ productCollection.count } products');

[...productCollection].forEach(p =>
    console.log(`Product: ${p.name}, ${ p.price}`));
```

Новое свойство реализует интерфейс `Iterable<T>`, что указывает на то, что оно определяет свойство `Symbol.iterator`, возвращающее объект `Iterator<T>`, который может быть использован для итерации. Код в листинге 13.7 выдает следующий результат:

```
There are 2 products
Product: Running Shoes, 100
Product: Hat, 25
```

13.4. ИНДЕКСНЫЕ ТИПЫ

Класс `Collection<T>` имеет ограничения на принимаемые им типы благодаря структуре типа. Это гарантирует, что все объекты, с которыми он работает, имеют свойство `name`, которое можно использовать в качестве ключа для хранения и извлечения объектов в `Map`.

TypeScript предоставляет набор связанных функций, позволяющих использовать в роли ключа любое свойство, определенное объектом, сохраняя при этом типобезопасность. Такие возможности сложны для понимания, поэтому мы рассмотрим, как они работают изолированно, а затем используем их для улучшения класса `Collection<T>`.

13.4.1. Запрос индексного типа

Ключевое слово `keyof`, также известное как оператор запроса индексного типа, возвращает объединение имен свойств типа с помощью функциональности типа с литеральными значениями, описанной в главе 9. В листинге 13.8 показано применение `keyof` к классу `Product`.

Листинг 13.8. Использование оператора запроса индексного типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.ts";

let myVar: keyof Product = "name";
myVar = "price";
myVar = "someOtherName";
```

Аннотацией типа для переменной `myVar` является `keyof Product`, которая будет объединением имен свойств, определенных классом `Product`. Следовательно, переменной `myVar` могут быть присвоены только строковые значения `name` и `price`, поскольку это имена единственных двух свойств, определенных классом `Product` в файле `dataTypes.ts`, созданном в главе 12:

```
...
export class Product {
    constructor(public name: string, public price: number) {}
}
...
```

Попытка присвоить любое другое значение переменной `myVar`, как это делает заключительный оператор в листинге 13.8, приведет к ошибке компилятора:

```
src/index.ts(5,1): error TS2322: Type '"someOtherName"' is not assignable
to type 'keyof Product'.
```

Ключевое слово `keyof` можно использовать для ограничения параметров обобщенного типа, чтобы они могли иметь только тип, соответствующий свойствам другого типа, как показано в листинге 13.9.

Листинг 13.9. Ограничение параметра обобщенного типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

function getValue<T, K extends keyof T>(item: T, keyname: K) {
    console.log('Value: ${item[keyname]}');
}

let p = new Product("Running Shoes", 100);
getValue(p, "name");
getValue(p, "price");

let e = new Employee("Bob Smith", "Sales");
getValue(e, "name");
getValue(e, "role");
```

В данном примере определена функция `getValue`, у которой параметр типа `K` ограничен с помощью `typeof T`. Это означает, что `K` может быть именем только одного из свойств, определяемых `T`, независимо от типа, используемого для `T` при вызове функции. Когда функция `getValue` используется с объектом `Product`, параметр `keyname` может быть только `name` или `price`. А когда функция `getValue` используется с объектом `Employee`, параметр `keyname` может быть только `name` или `role`. В обоих случаях параметр `keyname` может быть использован для безопасного получения или установки значения соответствующего свойства из объекта `Product` или `Employee`. Код в листинге 13.9 выдает следующий результат:

```
Value: Running Shoes
Value: 100
Value: Bob Smith
Value: Sales
```

13.4.2. Явное указание параметров обобщенных типов для индексных типов

В листинге 13.9 метод `getValue` был вызван без указания аргументов обобщенного типа, что позволило компилятору вывести типы из аргументов функции. Явное указание аргументов типа раскрывает один из аспектов использования оператора запроса индексного типа, который может сбить с толку (листинг 13.10).

Листинг 13.10. Использование явных аргументов типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

function getValue<T, K extends keyof T>(item: T, keyname: K) {
    console.log('Value: ${item[keyname]}');
}
```

```
let p = new Product("Running Shoes", 100);
getValue<Product, "name">(p, "name");
getValue(p, "price");

let e = new Employee("Bob Smith", "Sales");
getValue(e, "name");
getValue(e, "role");
```

Может показаться, что свойство, необходимое для примера, указано дважды, но `name` имеет два разных назначения в модифицированном операторе, как показано на рис. 13.1.



Рис. 13.1. Тип и значение индекса

В качестве аргумента обобщенного типа `name` представляет собой тип с литеральным значением, который указывает на один из типов `Keyof Product` и используется компилятором TypeScript для проверки типов. А в роли аргумента функции `name` представляет собой значение типа `string`, которое используется средой выполнения JavaScript при выполнении кода. Код в листинге 13.10 выдает следующий результат:

```
Value: Running Shoes
Value: 100
Value: Bob Smith
Value: Sales
```

13.4.3. Оператор индексированного доступа

Оператор индексированного доступа используется для извлечения типа одного или нескольких свойств, как показано в листинге 13.11.

Листинг 13.11. Использование оператора индексированного доступа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

function getValue<T, K extends keyof T>(item: T, keyname: K) {
    console.log('Value: ${item[keyname]}');
}

type priceType = Product["price"];
type allTypes = Product[keyof Product];

let p = new Product("Running Shoes", 100);
getValue<Product, "name">(p, "name");
getValue(p, "price");

let e = new Employee("Bob Smith", "Sales");
getValue(e, "name");
getValue(e, "role");
```

Оператор индексированного доступа выражается с помощью квадратных скобок после типа. Например, `Product["price"]` — это число, поскольку именно такой тип имеет свойство `price`, определяемое классом `Product`. Оператор индексированного доступа работает с литеральными типами значений, поэтому его можно использовать в запросах индексного типа, например, так:

```
...
type allTypes = Product[keyof Product];
...
```

Выражение `keyof Product` возвращает объединение типов литеральных значений с именами свойств, определенных в классе `Product`, — `"name" | "price"`. Оператор индексированного доступа возвращает объединение типов этих свойств, например `Product[keyof Product]` — `string | number`, что является объединением типов свойств `name` и `price`.

СОВЕТ

Типы, возвращаемые оператором индексированного доступа, называются *типами поиска*.

Оператор индексированного доступа чаще всего используется с обобщенными типами, что позволяет безопасно работать с типами свойств, даже если конкретные типы будут неизвестны (листинг 13.12).

Листинг 13.12. Использование оператора индексированного доступа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

function getValue<T, K extends keyof T>(item: T, keyname: K): T[K] {
    return item[keyname];
}

let p = new Product("Running Shoes", 100);
console.log(getValue<Product, "name">(p, "name"));
console.log(getValue(p, "price"));

let e = new Employee("Bob Smith", "Sales");
console.log(getValue(e, "name"));
console.log(getValue(e, "role"));
```

Оператор индексированного доступа выражается с помощью обычного типа, его типа `keyof` и квадратных скобок, как показано на рис. 13.2.

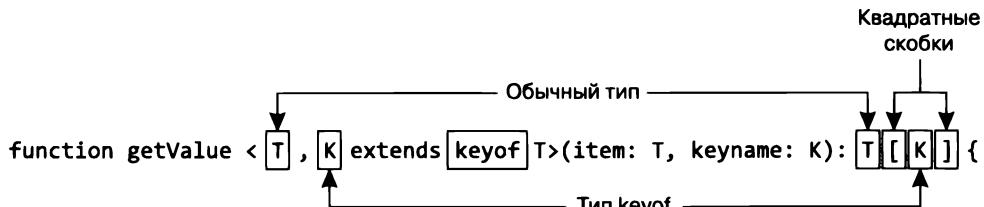


Рис. 13.2. Оператор индексированного доступа

Оператор индексированного доступа `T[K]` из листинга 13.12 сообщает компилятору, что результат функции `getValue` будет иметь тип свойства, имя которого указано в аргументе типа `keyof`. Это позволяет компилятору самому определить тип результата на основе аргументов обобщенного типа, используемых при вызове функции. Для объекта `Product` это означает, что аргумент `name` приведет к строковому результату, а `price` — к числовому. Код в листинге 13.12 выдает следующий результат:

```
Running Shoes
100
Bob Smith
Sales
```

13.4.4. Использование индексного типа для класса `collection<t>`

Использование индексного типа позволяет модифицировать класс `Collection<T>` таким образом, чтобы он мог хранить объекты любого типа, а не только те, которые определяют свойство `name`. В листинге 13.13 показаны изменения класса, в котором с помощью запроса индексного типа свойство конструктора `propertyName` ограничено именами свойств, определяемых параметром обобщенного типа `T`, обеспечивающим ключ, по которому объекты могут храниться в `Map`.

Листинг 13.13. Использование индексного типа в классе `collection` в файле `index.ts`

```
import { City, Person, Product, Employee } from "./dataTypes.js";

let products = [new Product("Running Shoes", 100), new Product("Hat", 25)];

//type shapeType = { name: string };

class Collection<T, K extends keyof T> implements Iterable<T> {
    private items: Map<T[K], T>;

    constructor(initialItems: T[] = [], private propertyName: K) {
        this.items = new Map<T[K], T>();
        this.add(...initialItems);
    }

    add(...newItems: T[]): void {
        newItems.forEach(newItem =>
            this.items.set(newItem[this.propertyName], newItem));
    }

    get(key: T[K]): T {
        return this.items.get(key);
    }

    get count(): number {
        return this.items.size;
    }
}
```

```

[Symbol.iterator](): Iterator<T> {
    return this.items.values();
}

let productCollection: Collection<Product, "name">
    = new Collection(products, "name");
console.log('There are ${ productCollection.count } products');

let itemByKey = productCollection.get("Hat");
console.log('Item: ${ itemByKey.name }, ${ itemByKey.price }');

```

Класс был переписан с дополнительным параметром обобщенного типа K, который ограничивается keyof T — типом данных объектов, хранящихся в коллекции. Новый экземпляр Collection<T, K> создается следующим образом:

```

...
let productCollection: Collection<Product, "name">
    = new Collection(products, "name");
...

```

Результат выполнения кода из листинга 13.13:

```

There are 2 products
Item: Hat, 25

```

Частые цепочки угловых и квадратных скобок в листинге 13.13 могут быть сложны для понимания, если вы только начинаете использовать индексные типы. Чтобы облегчить понимание кода, в табл. 13.5 описаны важные параметры типа, конструктора и соответствующие типы, в которые они разрешаются для объекта Collection<Product, "name">, созданного в примере.

Таблица 13.5. Типы, используемые классом Collection<T>

Название	Описание
T	Тип объектов, хранящихся в классе коллекции. Он предоставляется первым аргументом обобщенного типа, являющимся Product для создаваемого в листинге объекта
K	Имя ключевого свойства, которое ограничено именами свойств, определенных T. Значение для этого типа предоставляется вторым аргументом обобщенного типа, который является свойством name для объекта, создаваемого в листинге
T[K]	Тип свойства ключа, который получается с помощью оператора индексированного доступа и используется для указания типа ключа при создании объекта Map, а также для ограничения типа параметров. Это тип свойства Product.name для объекта, созданного в листинге, и он является string
propertyName	Имя ключевого свойства, которое требуется в качестве значения, доступного среди выполнения JavaScript после удаления информации об обобщенном типе TypeScript. Для объекта, созданного в примере, этим значением является name, соответствующее обобщенному типу K

В результате использования индексного типа в листинге 13.13 становится возможным хранение объектов с использованием любого свойства, а также сохранение объектов различных типов. В листинге 13.14 изменен способ создания класса `Collection<T, K>` таким образом, что теперь в качестве ключа используется свойство `price`. В листинге также опущены аргументы обобщенных типов, и компилятор может самостоятельно определить необходимые типы.

Листинг 13.14. Изменение ключевого свойства в файле index.ts

```
...
let productCollection = new Collection(products, "price");
console.log('There are ${ productCollection.count } products');

let itemByKey = productCollection.get(100);
console.log('Item: ${ itemByKey.name }, ${ itemByKey.price }');
...
```

Тип аргумента метода `get` меняется в соответствии с типом свойства ключа, что позволяет получать объекты с помощью аргумента `number`. Код в листинге 13.14 выдает следующий результат:

```
There are 2 products
Item: Running Shoes, 100
```

13.5. СОПОСТАВЛЕНИЕ ТИПА

Сопоставленные типы создаются путем применения преобразования к свойствам существующего типа. Лучший способ понять, как это работает, — создать тип, который обрабатывает тип, но не изменяет его (листинг 13.15).

Листинг 13.15. Использование сопоставленного типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type MappedProduct = {
  [P in keyof Product] : Product[P]
};

let p: MappedProduct = { name: "Kayak", price: 275};
console.log('Mapped type: ${p.name}, ${p.price}'');
```

Сопоставление типов представляет собой выражение, в котором выбираются имена свойств, включаемых в сопоставленный тип, и указывается тип для каждого из них, как показано на рис. 13.3.

```
type MappedProduct = {
  [P in keyof Product] : Product[P]
};
```

Рис. 13.3. Сопоставленный тип

Селектор имени свойства определяет параметр типа, названный в данном примере `P`, и использует ключевое слово `in` для перечисления типов в объединении литеральных значений. Объединение типов может быть выражено напрямую, например `"name" | "price"`, или получено с помощью `keyof`.

Компилятор TypeScript создает новое свойство в сопоставленном типе для всех типов в объединении. Тип каждого свойства определяется селектором типа, который можно получить из исходного типа с помощью оператора индексированного доступа с указанием `P` в качестве типа с литеральным значением для поиска.

Тип `MappedProduct` в листинге 13.15 использует `keyof` для выбора свойств, определенных классом `Product`, и оператор индексированного типа для получения типа каждого из этих свойств. Результат эквивалентен данному типу:

```
type MappedProduct = {  
    name: string;  
    price: number;  
}
```

Код в листинге 13.15 выдает следующий результат:

```
Mapped type: Kayak, 275
```

13.5.1. Изменение имен и типов сопоставления

В предыдущем примере в процессе сопоставления сохранялись имена и типы свойств. Однако сопоставление типов поддерживает изменение как имени, так и типа свойств в новом типе, как показано в листинге 13.16.

Листинг 13.16. Изменение имен и типов в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";  
  
type MappedProduct = {  
    [P in keyof Product] : Product[P]  
};  
  
let p: MappedProduct = { name: "Kayak", price: 275};  
console.log('Mapped type: ${p.name}, ${p.price}');  
  
type AllowStrings = {  
    [P in keyof Product] : Product[P] | string  
}  
let q: AllowStrings = { name: "Kayak", price: "apples" };  
console.log('Changed type # 1: ${q.name}, ${q.price}');  
  
type ChangeNames = {  
    [P in keyof Product as '${P}Property'] : Product[P]  
}  
  
let r: ChangeNames = { nameProperty: "Kayak", priceProperty: 12 };  
console.log('Changed type # 2: ${r.nameProperty}, ${r.priceProperty}');
```

Тип `AllowStrings` создается с помощью сопоставления, которое создает объединение типов между строкой и исходным типом свойства, например, так:

```
...
[P in keyof Product] : Product[P] | string
...
```

В результате получается тип, эквивалентный данному типу:

```
type AllowStrings = {
  name: string;
  price: number | string;
}
```

Тип `ChangeNames` создается с сопоставлением, которое изменяет имя каждого свойства путем добавления `Property`.

```
...
[P in keyof Product as `${P}Property`] : Product[P]
...
```

Ключевое слово `as` сочетается с выражением, определяющим имя свойства. В этом случае для модификации существующего имени используется строка-шаблон, результат которой эквивалентен следующему типу:

```
type ChangeNames = {
  nameProperty: string;
  priceProperty: number;
}
```

Код в листинге 13.16 при компиляции и выполнении выдаст следующий результат:

```
Mapped type: Kayak, 275
Changed type # 1: Kayak, apples
Changed type # 2: Kayak, 12
```

13.5.2. Параметр обобщенного типа с сопоставленным типом

Сопоставленные типы полезнее, когда они определяют параметр обобщенного типа (листинг 13.17), что позволяет применять описываемое ими преобразование к более широкому диапазону типов.

Листинг 13.17. Использование параметра обобщенного типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type Mapped<T> = {
  [P in keyof T] : T[P]
};

let p: Mapped<Product> = { name: "Kayak", price: 275};
console.log('Mapped type: ${p.name}, ${p.price}');
```

```
let c: Mapped<City> = { name: "London", population: 8136000};
console.log('Mapped type: ${c.name}, ${c.population}');
```

Тип `Mapped<T>` определяет параметр обобщенного типа `T`, который является типом, подлежащим преобразованию. Параметр типа используется в селекторах имени и типа, то есть любой тип может быть сопоставлен с помощью параметра обобщенного типа. В листинге 13.17 сопоставленный тип `Mapped<T>` используется для классов `Product` и `City` и выводит следующий результат:

```
Mapped type: Kayak, 275
Mapped type: London, 8136000
```

13.5.3. Изменение обязательности и изменчивости свойств

Сопоставленные типы могут изменять свойства, делая их необязательными или обязательными, а также добавлять или удалять ключевое слово `readonly`, как показано в листинге 13.18.

Листинг 13.18. Изменение свойств в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type MakeOptional<T> = {
    [P in keyof T]? : T[P]
};

type MakeRequired<T> = {
    [P in keyof T]-? : T[P]
};

type MakeReadOnly<T> = {
    readonly [P in keyof T] : T[P]
};

type MakeReadWrite<T> = {
    -readonly [P in keyof T] : T[P]
};

type optionalType = MakeOptional<Product>;
type requiredType = MakeRequired<optionalType>;
type readOnlyType = MakeReadOnly<requiredType>;
type readWriteType = MakeReadWrite<readOnlyType>;

let p: readWriteType = { name: "Kayak", price: 275};
console.log('Mapped type: ${p.name}, ${p.price}');
```

Чтобы сделать свойства сопоставленного типа необязательными, после селектора имени ставится символ `?`, а символы `-?` используются для того, чтобы сделать свойства обязательными. Свойствам присваивается статус «только для чтения»

и «для чтения и записи», для чего перед селектором имен пишутся ключевые слова `readonly` и `-readonly` соответственно.

Сопоставленные типы изменяют все свойства, определяемые исходным типом, так что, например, тип, полученный с помощью `MakeOptional<T>` при применении к классу `Product`, эквивалентен типу:

```
type optionalType = {
  name?: string;
  price?: number;
}
```

Типы, созданные сопоставлениями, могут быть переданы другим сопоставлениям, создавая таким образом цепочку преобразований. В листинге тип, полученный с помощью сопоставления `MakeOptional<T>`, затем преобразуется сопоставлением `MakeRequired<T>`, результата которого подается на вход сопоставлению `MakeReadOnly<T>`, а затем сопоставлению `MakeReadWrite<T>`. В итоге свойства сначала делаются необязательными, затем обязательными, потом только для чтения и, наконец, доступными для записи. Код из листинга 13.18 выдает следующее:

```
Mapped type: Kayak, 275
```

13.5.4. Основные встроенные сопоставления

TypeScript предоставляет встроенные сопоставленные типы, некоторые из которых соответствуют преобразованиям, приведенным в листинге 13.18, а некоторые описаны в последующих разделах. В табл. 13.6 описаны основные встроенные отображения.

Таблица 13.6. Основные сопоставления типов

Название	Описание
<code>Partial<T></code>	Делает все свойства в типе необязательными
<code>Required<T></code>	Делает все свойства в типе обязательными
<code>Readonly<T></code>	Добавляет в свойства ключевое слово <code>readonly</code>
<code>Pick<T, K></code>	Создает новый тип, выбирая только указанные свойства <code>K</code> из исходного типа <code>T</code>
<code>Omit<T, keys></code>	Создает новый тип, исключая указанные свойства <code>keys</code> из исходного типа <code>T</code>
<code>Record<T, K></code>	Создает новый тип, где ключами будут ключи из типа <code>K</code> , а значениями — значения из типа <code>T</code>

Встроенной команды для удаления ключевого слова `readonly` не существует, но в листинге 13.19 сопоставления заменены теми, которые предоставляет TypeScript.

Листинг 13.19. Использование встроенных команд сопоставления в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

// type MakeOptional<T> = {
//   [P in keyof T]?: T[P]
// };

// type MakeRequired<T> = {
//   [P in keyof T]-?: T[P]
// };

// type MakeReadOnly<T> = {
//   readonly [P in keyof T] : T[P]
// };

type MakeReadWrite<T> = {
  -readonly [P in keyof T] : T[P]
};

type optionalType = Partial<Product>;
type requiredType = Required<optionalType>;
type readOnlyType = Readonly<requiredType>;
type readWriteType = MakeReadWrite<readOnlyType>;
let p: readWriteType = { name: "Kayak", price: 275};
console.log('Mapped type: ${p.name}, ${p.price}'');
```

Сопоставления из табл. 13.6 имеют тот же эффект, что и определенные в листинге 13.19, и код в листинге 13.19 выдает следующий результат:

Mapped type: Kayak, 275

Сопоставление определенных свойств

Запрос индексного типа для сопоставленного типа можно выразить в виде параметра обобщенного типа, который затем допускается использовать для выбора конкретных свойств для сопоставления по имени, как показано в листинге 13.20.

Листинг 13.20. Выбор определенных свойств в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type SelectProperties<T, K extends keyof T> = {
  [P in K]: T[P]
};

let p1: SelectProperties<Product, "name"> = { name: "Kayak" };
let p2: Pick<Product, "name"> = { name: "Kayak" };
let p3: Omit<Product, "price"> = { name: "Kayak" };
console.log('Custom mapped type: ${p1.name}');
console.log('Built-in mapped type (Pick): ${p2.name}');
console.log('Built-in mapped type (Omit): ${p3.name}'');
```

Сопоставление `SelectProperties` определяет дополнительный параметр обобщенного типа `K`, который ограничен ключевым словом `keyof` таким образом, чтобы можно было указать только типы, соответствующие свойствам, определенным

параметром типа `T`. Параметр нового типа используется в селекторе имен сопоставления, что позволяет выбирать отдельные свойства для включения в сопоставленный тип, например, так:

```
...
let p1: SelectProperties<Product, "name"> = { name: "Kayak" };
...
```

Это сопоставление выбирает свойство `name`, определенное классом `Product`. Несколько свойств можно выразить в виде объединения типов, и TypeScript предоставляет встроенное сопоставление `Pick<T, K>`, выполняющее ту же функцию.

```
...
let p2: Pick<Product, "name"> = { name: "Kayak" };
...
```

Сопоставление `Pick` указывает ключи, которые должны храниться в сопоставленном типе. Сопоставление `Omit` работает противоположным образом и исключает один или несколько ключей.

```
...
let p3: Omit<Product, "price"> = { name: "Kayak" };
...
```

Результат всех трех сопоставлений одинаков, и код из листинга 13.20 выдает следующий результат:

```
Custom mapped type: Kayak
Built-in mapped type (Pick): Kayak
Built-in mapped type (Omit): Kayak
```

13.5.5. Комбинирование преобразований в одном сопоставлении

В листинге 13.19 показано, как можно комбинировать сопоставления для создания цепочки преобразований. Но при этом сопоставления могут применять несколько изменений к свойствам, как показано в листинге 13.21.

Листинг 13.21. Комбинирование преобразований в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type CustomMapped<T, K extends keyof T> = {
    readonly[P in K]?: T[P]
};

type BuiltInMapped<T, K extends keyof T> = Readonly<Partial<Pick<T, K>>>;
```

```
let p1: CustomMapped<Product, "name"> = { name: "Kayak" };
let p2: BuiltInMapped<Product, "name" | "price">
    = { name: "Lifejacket", price: 48.95};
console.log('Custom mapped type: ${p1.name}');
console.log('Built-in mapped type: ${p2.name}, ${p2.price}'');
```

Для сопоставлений пользовательских типов в одном преобразовании допускается использовать вопросительный знак (?) и ключевое слово `readonly`, которое может быть ограничено, чтобы разрешить выбор свойства по имени. Сопоставления

также могут быть объединены в цепочку, как показано на примере комбинации сопоставлений `Pick`, `partial` и `Readonly`. Код, приведенный в листинге 13.21, выдает следующий результат:

```
Custom mapped type: Kayak
Built-in mapped type: Lifejacket, 48.95
```

13.5.6. Создание типов с помощью сопоставления типов

Последней особенностью, предоставляемой сопоставлениями типов, является возможность создавать новые типы, а не просто преобразовывать определенные. В листинге 13.22 показано базовое использование этой функции: создается тип, содержащий свойства `name` и `city`.

Листинг 13.22. Создание типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type CustomMapped<K extends keyof any, T> = {
  [P in K]: T
};

let p1: CustomMapped<"name" | "city", string>
= { name: "Bob", city: "London"};
let p2: Record<"name" | "city", string> = { name: "Alice", city: "Paris"};

console.log('Custom mapped type: ${p1.name}, ${p1.city}');
console.log('Built-in mapped type: ${p2.name}, ${p2.city}');
```

Первый параметр обобщенного типа ограничен с помощью `keyof any`. Это означает, что можно указать объединение типов с литеральным значением, которое может содержать имена свойств, необходимых для нового типа. Второй параметр обобщенного типа используется для задания типа создаваемых свойств и применяется следующим образом:

```
...
let p1: CustomMapped<"name" | "city", string>
= { name: "Bob", city: "London"};
...
```

В результате сопоставления получается тип с двумя строковыми свойствами: `name` и `city`. TypeScript предоставляет встроенное сопоставление `Record`, которое выполняет ту же задачу.

```
...
let p2: Record<"name" | "city", string> = { name: "Alice", city: "Paris"};
...
```

Эту функцию в своих проектах я использую реже всего, но мне хотелось показать, что сопоставления более гибкие, чем может показаться на первый взгляд, и что типы с литеральными значениями, ограниченные с помощью `keyof any`, могут принимать любую комбинацию имен свойств. Код, приведенный в листинге 13.22, выдает следующий результат:

```
Custom mapped type: Bob, London
Built-in mapped type: Alice, Paris
```

13.6. УСЛОВНЫЕ ТИПЫ

Условные типы — это выражения, содержащие параметры обобщенного типа, которые оцениваются для выбора новых типов. В листинге 13.23 показан пример базового условного типа.

Листинг 13.23. Использование условного типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type resultType<T extends boolean> = T extends true ? string : number;

let firstVal: resultType<true> = "String Value";
let secondVal: resultType<false> = 100;

let mismatchCheck: resultType<false> = "String Value";
```

Условные типы имеют параметр обобщенного типа и тернарное выражение, которое выбирает тип результата, как показано на рис. 13.4.

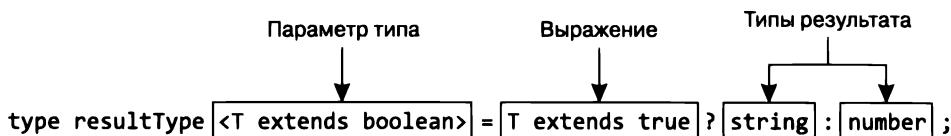


Рис. 13.4. Условный тип

Условный тип представляет собой заполнитель для одного из его типов результатов, который не выбирается до тех пор, пока не будет использован параметр обобщенного типа, который позволяет оценивать выражение, используя один из выбранных типов результатов.

В листинге условный тип `resultType<T>` является заполнителем для типов `string` и `number`, то есть аргумент для обобщенного типа `T` будет определять, к какому типу относится условный тип — `string` или `number`. Параметр обобщенного типа `T` ограничен таким образом, что он может принимать только логические значения, и выражение будет оцениваться как `true`, если аргумент, указанный для `T`, является типом с литеральным значением `true`. В результате, когда `T` равно `true`, `resultType<T>` разрешается как `string`:

```
...
let firstVal: resultType<true> = "String Value";
let stringTypeCheck: string = firstVal;
...
```

Компилятор определяет условный тип и знает, что аннотация типа для `firstVal` разрешается в `string`, что позволяет присвоить `firstVal` строковое литеральное значение. Когда аргумент обобщенного типа равен `false`, условный тип разрешается в `number`:

```
...
let secondVal: resultType<false> = 100;
let numberTypeCheck: number = secondVal;
...
```

Компилятор обеспечивает безопасность типов с помощью условных типов. В последнем утверждении листинга 13.23 условный тип разрешается как `number`, но ему присваивается значение `string`, что приводит к следующей ошибке компилятора: `error TS2322: Type 'string' is not assignable to type 'number'`.

РИСКИ УСЛОВНЫХ ТИПОВ

Условные типы — это продвинутая функция, которую следует использовать с осторожностью. Написание условных типов — довольно сложный процесс, и часто кажется какой-то магией, когда вы проводите компилятор через ряд выражений, чтобы получить нужные результаты.

С увеличением сложности условного типа возрастает риск того, что вы не сможете правильно учесть все варианты перестановки типов и получите слишком нестрогий результат, создающий дыру при проверке типов, или слишком строгий, приводящий к ошибкам компилятора при корректном использовании.

При использовании условных типов следует помнить, что вы лишь описываете комбинации типов компилятору TypeScript, а информация о типах будет удалена в процессе компиляции. И, когда условный тип усложняется и охватывает все больше комбинаций, следует задуматься о том, нет ли более простого способа достичь того же результата.

13.6.1. Вложенные условные типы

Более сложные комбинации типов можно описать с помощью вложенных условных типов. Результатом условного типа может быть другой условный тип, и компилятор будет следовать по цепочке выражений до тех пор, пока не достигнет результата, не являющегося условным, как показано в листинге 13.24.

Листинг 13.24. Вложенные условные типы в файле `index.ts` из папки `src`

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type resultType<T extends boolean> = T extends true ? string : number;

type references = "London" | "Bob" | "Kayak";

type nestedType<T extends references>
    = T extends "London" ? City : T extends "Bob" ? Person : Product;

let firstVal: nestedType<"London"> = new City("London", 8136000);
let secondVal: nestedType<"Bob"> = new Person("Bob", "London");
let thirdVal: nestedType<"Kayak"> = new Product("Kayak", 275);
```

Тип `nestedType<T>` представляет собой вложенный условный тип, предназначенный для выбора между тремя типами результатов в зависимости от значения параметра обобщенного типа. Как отмечалось во врезке, сложные условные типы трудно понять, и это особенно актуально, когда они вложены друг в друга.

13.6.2. Условные типы в обобщенных классах

Условные типы могут быть использованы для выражения взаимосвязи между типами параметров метода или функции и их результатами, как показано в листинге 13.25. Это более лаконичная альтернатива перегрузке типов функций, описанной в главе 8, хотя условные типы могут быть более сложными для понимания.

Листинг 13.25. Определение обобщенного типа в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type resultType<T extends boolean> = T extends true ? string : number;

class Collection<T> {
    private items: T[];

    constructor(...initialItems: T[]) {
        this.items = initialItems || [];
    }

    total<P extends keyof T, U extends boolean>(propName: P, format: U)
        : resultType<U> {
        let totalValue = this.items.reduce((t, item) =>
            t += Number(item[propName]), 0);
        return format ? `${totalValue.toFixed()}` : totalValue as any;
    }
}

let data = new Collection<Product>(new Product("Kayak", 275),
    new Product("Lifejacket", 48.95));

let firstVal: string = data.total("price", true);
console.log('Formatted value: ${firstVal}');
let secondVal: number = data.total("price", false);
console.log('Unformatted value: ${secondVal}'');
```

Класс `Collection<T>` использует массив для хранения объектов, тип которых задается параметром обобщенного типа с именем `T`. Метод `total` определяет два параметра обобщенного типа: `P`, устанавливающий свойство для создания итога, и `U`, который указывает, должен ли результат быть отформатирован. Результатом работы метода `total` является условный тип, который разрешается по значению, указанному в параметре типа `U`:

```
...
total<P extends keyof T, U extends boolean>(propName: P, format: U)
    : resultType<U> {
...
}
```

Использование условного типа означает, что результат работы метода `total` определяется аргументом, предоставленным для параметра типа `U`. А поскольку компилятор может вывести `U` из значения, предоставленного для аргумента `format`, как объяснялось в главе 12, метод можно вызвать следующим образом:

```
...
let firstVal: string = data.total("price", true);
...
```

Если аргумент параметра `format` равен `true`, то условный тип разрешается так, что типом результата метода `total` становится `string`. Это соответствует типу данных, возвращаемому реализацией метода:

```
...
return format ? `${totalValue.toFixed()}` : totalValue as any;
...
```

Если аргумент параметра `format` равен `false`, то условный тип разрешается так, что типом метода `total` становится `number`. Это позволяет методу возвращать неформатированное числовое значение `value`:

```
...
return format ? `${totalValue.toFixed()}` : totalValue as any;
...
```

ВОЗВРАТ ЗНАЧЕНИЙ В МЕТОДАХ, ИСПОЛЬЗУЮЩИХ УСЛОВНЫЙ ТИП

На момент написания книги компилятор TypeScript испытывает трудности с сопоставлением типа данных значений, возвращаемых методами и функциями при использовании условных типов. Именно по этой причине в листинге 13.25 в методе `total` используется утверждение типа, указывающее компилятору, что результат должен рассматриваться как `any`. Без указания типа компилятор выдаст ошибку.

Код в листинге 13.25 выдает следующий результат:

```
Formatted value: $324
Unformatted value: 323.95
```

13.6.3. Использование условных типов в объединениях типов

Условные типы можно использовать для фильтрации объединений типов, что позволяет легко выбирать или исключать типы из набора, содержащегося в объединении (листинг 13.26).

Листинг 13.26. Фильтрация объединения типов в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type Filter<T, U> = T extends U ? never : T;

function FilterArray<T, U>(data: T[], predicate: (item) => item is U): Filter<T, U>[] {
    return data.filter(item => !predicate(item)) as any;
}
```

```

let dataArray = [new Product("Kayak", 275), new Person("Bob", "London"),
    new Product("Lifejacket", 27.50)];

function isProduct(item: any): item is Product {
    return item instanceof Product;
}

let filteredData: Person[] = FilterArray(dataArray, isProduct);
filteredData.forEach(item => console.log('Person: ${item.name}'));

```

Когда условный тип используется в объединении типов, компилятор TypeScript распределяет условие по каждому типу в объединении, создавая так называемый *распределительный условный тип*. Такой эффект наблюдается, когда условный тип используется как объединение типов, например, так:

```

...
type filteredUnion = Filter<Product | Person, Product>
...

```

Компилятор TypeScript применяет условный тип к каждому типу в объединении отдельно, а затем создает объединение результатов, как показано ниже:

```

...
type filteredUnion = Filter<Product, Product> | Filter<Person, Product>
...

```

Условный тип `Filter<T, U>` оценивается как `never`, когда первый параметр типа совпадает со вторым, что приводит к такому результату:

```

...
type filteredUnion = never | Person
...

```

Объединение с `never` создать невозможно, поэтому компилятор исключает его из объединения, в результате чего `Filter<Product | Person, Product>` эквивалентен этому типу:

```

...
type filteredUnion = Person
...

```

Условный тип отфильтровывает все типы, которые нельзя присвоить `Person`, и возвращает оставшиеся типы в объединении. Метод `FilterArray<T, U>` фильтрует массив с помощью предикатной функции и возвращает тип `Filter<T, U>`. Код в листинге 13.26 приводит к следующему результату:

```
Person: Bob
```

Использование встроенных распределительных условных типов

TypeScript предоставляет набор встроенных условных типов, предназначенных для фильтрации объединений (табл. 13.7), что позволяет выполнять общие задачи без необходимости определения пользовательских типов.

Таблица 13.7. Встроенные распределительные условные типы

Название	Описание
<code>Exclude<T, U></code>	Исключает типы, которые могут быть присвоены <code>U</code> из <code>T</code> , что эквивалентно типу <code>Filter<T, U></code> в листинге 13.26
<code>Extract<T, U></code>	Выбирает типы, которые могут быть назначены для <code>U</code> от <code>T</code>
<code>NonNullable<T></code>	Исключает из <code>T</code> значения <code>null</code> и <code>undefined</code>

13.6.4. Использование условных типов в сопоставлениях типов

Условные типы можно комбинировать с сопоставлениями типов. Это позволяет применять различные преобразования к свойствам типа, что обеспечивает большую гибкость, чем при использовании каждой из этих функций по отдельности. В листинге 13.27 показано сопоставление типов, использующее условный тип.

Листинг 13.27. Сопоставление и условный тип в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type changeProps<T, U, V> = {
    [P in keyof T]: T[P] extends U ? V : T[P]
};

type modifiedProduct = changeProps<Product, number, string>;

function convertProduct(p: Product): modifiedProduct {
    return { name: p.name, price: '$${p.price.toFixed(2)}' };
}

let kayak = convertProduct(new Product("Kayak", 275));
console.log('Product: ${kayak.name}, ${kayak.price}');
```

Сопоставление `changeProps<T, U, V>` выбирает свойства типа `U` и преобразует их в тип `V` в сопоставленном типе. Данное выражение применяет сопоставление к классу `Product`, указывая, что свойства `number` должны быть превращены в свойства `string`:

```
...
type modifiedProduct = changeProp<Product, number, string>;
...
```

Сопоставленный тип определяет свойства `name` и `price`, которые имеют тип `string`. Тип `modifiedProduct` используется в качестве результата функции `convertProduct`, которая принимает объект `Product` и возвращает объект, соответствующий структуре сопоставленного типа за счет форматирования свойства `price`. Код, приведенный в листинге 13.27, выдает следующий результат:

```
Product: Kayak, $275.00
```

13.6.5. Определение свойств конкретного типа

Часто требуется ограничить параметр типа таким образом, чтобы его можно было использовать только для задания свойства, имеющего определенный тип. Например, в классе `Collection<T>` из листинга 13.25 определен метод `total`, который принимает свойство `name` и должен быть ограничен числовыми свойствами. Подобного ограничения можно достичь, объединив функции, описанные в предыдущих разделах, как показано в листинге 13.28.

Листинг 13.28. Определение свойств в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type unionOfTypeNames<T, U> = {
    [P in keyof T] : T[P] extends U ? P : never;
};

type propertiesOfType<T, U> = unionOfTypeNames<T, U>[keyof T];

function total<T, P extends propertiesOfType<T, number>>(data: T[],
    propName: P): number {
    return data.reduce((t, item) => t += Number(item[propName]), 0);
}

let products = [new Product("Kayak", 275),
    new Product("Lifejacket", 48.95)];
console.log('Total: ${total(products, "price")}'');
```

Метод выявления свойств нестандартен, поэтому мы разобьем процесс на два утверждения, чтобы было понятнее. Первый шаг — использовать сопоставления типов с условным оператором.

```
...
type unionOfTypeNames<T, U> = {
    [P in keyof T] : T[P] extends U ? P : never;
};
```

Условный оператор проверяет тип каждого свойства. Если свойство не имеет целевого типа, то его тип изменяется на `never`. Если у свойства ожидаемый тип, то его тип заменяется на литеральное значение, которое является именем свойства. Это означает, что сопоставление `unionOfTypeNames<Product, number>` дает следующий сопоставленный тип:

```
...
{
    name: never,
    price: "price"
}
```

Этот необычный сопоставленный тип предоставляет входные данные для второго этапа процесса, который заключается в использовании оператора

индексированного доступа для получения объединения типов свойств, определяемых сопоставленным типом:

```
...
type propertiesOfType<T, U> = unionOfTypeNames<T, U>[keyof T];
...
```

Для сопоставленного типа, созданного с помощью `unionOfTypeNames<Product, number>`, оператор индексированного доступа приводит к следующему объединению:

```
...
never | "price"
...
```

Как уже отмечалось, `never` автоматически исключается из объединений, оставляя лишь объединение типов с лiteralными значениями, которые представляют собой имена свойств требуемого типа. Это объединение имен свойств допускается использовать для ограничения параметров обобщенного типа.

```
...
function total<T, P extends propertiesOfType<T, number>>(data: T[],
    propName: P): number {
    return data.reduce((t, item) => t += Number(item[propName]), 0);
}
...
```

Параметр `propName` функции `total` может быть использован только с именами свойств `number` в типе `T`, как, например, здесь:

```
...
console.log('Total: ${total(products, "price")}');
...
```

Данный пример показывает, насколько гибкими бывают обобщенные типы в TypeScript, а также демонстрирует, что для достижения определенного эффекта могут потребоваться необычные шаги. Код в листинге 13.28 выдает следующий результат:

```
Total: 323.95
```

13.6.6. Вывод дополнительных типов в условиях

Может возникнуть противоречие между необходимостью принимать широкий диапазон типов через параметр обобщенного типа и необходимостью знать детали этих типов. В качестве примера в листинге 13.29 показана функция, принимающая массив или одиночный объект заданного типа.

Листинг 13.29. Определение функции в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

function getValue<T, P extends keyof T>(data: T, propName: P): T[P] {
    if (Array.isArray(data)) {
```

```

        return data[0][propName];
    } else {
        return data[propName];
    }
}

let products = [new Product("Kayak", 275),
    new Product("Lifejacket", 48.95)];
console.log('Array Value: ${getValue(products, "price")}');
console.log('Single Total: ${getValue(products[0], "price")}');

```

Этот код не скомпилируется, так как обобщенные параметры некорректно отражают связь между типами. Если функция `total` получает массив через параметр `data`, она возвращает значение свойства, указанного параметром `propName`, для первого элемента массива. Если же функция получает через `data` одиночный объект, она возвращает значение `propName` для этого объекта. Параметр `propName` ограничен с помощью `keyof`, что является проблемой при использовании массива, поскольку `keyof` возвращает объединение имен свойств, определенных объектом JavaScript-массива, а не свойства типа, содержащегося в массиве, что видно по сообщению об ошибке компилятора:

```
src/index.ts(13,48): error TS2345: Argument of type '"price"' is not assignable to parameter of type 'keyof Product[]'.
```

Ключевое слово `infer` в TypeScript позволяет выводить типы, которые неявно указаны в параметрах условного типа. Для нашего примера это означает, что мы можем попросить компилятор вывести тип объектов в массиве, как показано в листинге 13.30.

Листинг 13.30. Вывод типа массива в файле index.ts

```

import { City, Person, Product, Employee} from "./dataTypes.js";

type targetKeys<T> = T extends (infer U)[] ? keyof U: keyof T;

function getValue<T, P extends targetKeys<T>>(data: T, propName: P): T[P] {
    if (Array.isArray(data)) {
        return data[0][propName];
    } else {
        return data[propName];
    }
}

let products = [new Product("Kayak", 275),
    new Product("Lifejacket", 48.95)];
console.log('Array Value: ${getValue(products, "price")}');
console.log('Single Total: ${getValue(products[0], "price")}');

```

Типы выводятся с помощью ключевого слова `infer` и представляют собой обобщенный тип, который будет выведен компилятором, когда условный тип разрешится (рис. 13.5).

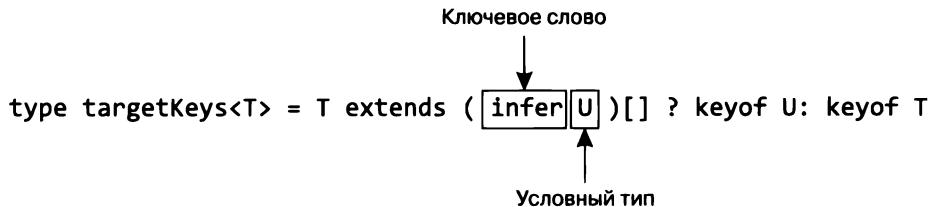


Рис. 13.5. Вывод типа в условном типе

В листинге 13.30 тип `U` выводится, если `T` — массив. Тип `U` выводится компилятором из параметра обобщенного типа `T` при разрешении типа. В результате типы `targetKeys<Product>` и `targetKeys<Product[]>` создают объединение `"name" | "price"`. Условный тип может быть использован для ограничения свойства функции `getValue<T, P>`, обеспечивая согласованную типизацию как для отдельных объектов, так и для массивов. Код из листинга 13.30 выдает следующий результат:

```
Array Value: 275
Single Total: 275
```

Вывод типов функций

Компилятор также может выводить типы в обобщенных типах, принимающих функции, как показано в листинге 13.31.

Листинг 13.31. Использование вывода типа для функции в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

type Result<T> = T extends (...args: any) => infer R ? R : never;

function processArray<T,
  Func extends (T) => any>(data: T[], func: Func): Result<Func>[] {
  return data.map(item => func(item));
}

let selectName = (p: Product) => p.name;

let products = [new Product("Kayak", 275),
  new Product("Lifejacket", 48.95)];
let names: string[] = processArray(products, selectName);
names.forEach(name => console.log('Name: ${name}'));
```

Условный тип `Result<T>` использует ключевое слово `infer` для получения типа результата функции, принимающей объект типа `T` и возвращающей результат типа `any`. Вывод типа позволяет использовать функции, обрабатывающие определенный тип, и при этом гарантировать, что результат функции `processArray` будет конкретного типа, исходя из результата функции, указанного в параметре `func`. Функция `selectName` возвращает строковое значение свойства `name` объекта `Product`, и вывод означает, что `Result<(...args: Product) => string>` правильно

идентифицирован как `string`. Это позволяет функции `processArray` вернуть результат типа `string[]`. Код в листинге 13.31 выдает следующий результат:

```
Name: Kayak
Name: Lifejacket
```

Вывод типов в условных типах — непростая задача, поэтому TypeScript предоставляет ряд встроенных условных типов, полезных при работе с функциями (табл. 13.8).

Таблица 13.8. Встроенные условные типы с выводом

Название	Описание
<code>Parameters<T></code>	Возвращает кортеж из типов всех параметров функции T
<code>ReturnType<T></code>	Возвращает тип результата функции T (эквивалент <code>Result<T></code> из листинга 13.31)
<code>ConstructorParameters<T></code>	Возвращает кортеж из типов всех параметров функции-конструктора
<code>InstanceType<T></code>	Возвращает тип результата функции-конструктора

Условные типы `ConstructorParameters<T>` и `InstanceType<T>` работают с функциями-конструкторами и наиболее полезны, когда нужно описать типы функций, создающих объекты, тип которых указан в качестве параметра обобщенного типа, как показано в листинге 13.32.

Листинг 13.32. Использование встроенных условных типов в файле index.ts

```
import { City, Person, Product, Employee } from "./dataTypes.js";

function makeObject<T extends new (...args: any) => any>
    (constructor: T, ...args: ConstructorParameters<T>)
    : InstanceType<T> {
    return new constructor(...args as any[]);
}

let prod: Product = makeObject(Product, "Kayak", 275);
let city: City = makeObject(City, "London", 8136000);

[prod, city].forEach(item => console.log(`Name: ${item.name}`));
```

Функция `makeObject` создает объекты из классов без знания дополнительной информации о том, какой класс требуется. Условные типы `ConstructorParameters<T>` и `InstanceType<T>` выводят параметры и результат для конструктора класса, указанного в качестве первого параметра обобщенного типа. Это гарантирует, что функция `makeObject` получит правильные типы для создания объекта, тип которого точно отражает тип созданного объекта. Код в листинге 13.32 выдает следующий результат:

```
Name: Kayak
Name: London
```

РЕЗЮМЕ

В этой главе вы познакомились с более сложными возможностями обобщенных типов TypeScript. Нет необходимости применять их в каждом проекте, однако их пользу трудно переоценить, особенно когда базовых возможностей недостаточно для описания типов, необходимых приложению.

- TypeScript поддерживает коллекции JavaScript с параметрами обобщенного типа и предоставляет итераторы, обеспечивающие безопасность типов.
- Индексные типы позволяют использовать свойство объекта в качестве ключа.
- Сопоставленные типы преобразуют свойства существующего типа. TypeScript предоставляет набор встроенных преобразований для создания сопоставленных типов.
- Условные типы – это выражения, оцениваемые для выбора одного типа на основе другого и обеспечивающие детальный контроль над обобщенными типами.

В следующей главе мы поговорим о декораторах, которые позволяют трансформировать поведение функций класса, таких как методы и свойства, не меняя их реализаций.

Декораторы

В этой главе

- ✓ Определение и применение декораторов.
- ✓ Декорирование классов, методов, свойств, аксессоров и автоаксессоров.
- ✓ Использование контекстных данных декораторов.
- ✓ Создание декораторов с помощью фабричной функции.
- ✓ Накопление данных о состоянии в декораторах.

Декораторы – это готовящееся дополнение к языку JavaScript, которое преобразует функции, определяемые классами. TypeScript давно поддерживает экспериментальную версию декораторов, в основном используемую при разработке на Angular. Однако в TypeScript 5 добавлена поддержка версии декораторов, которая будет принята в будущем выпуске спецификации JavaScript. В табл. 14.1 приведено краткое содержание главы.

Таблица 14.1. Краткое содержание главы

Задача	Решение	Листинг
Преобразовать функции класса	Определите и примените декоратор	9–12, 16–30, 38–41
Получить подробную информацию о функции, которая должна быть преобразована	Используйте контекстный объект декоратора	13–15

Задача	Решение	Листинг
Настроить каждое применение декоратора	Используйте фабричную функцию	31–37
Выполните начальную настройку декоратора	Используйте функцию-инициализатор	42–44
Накопить данные состояния	Определите переменную вне функции декоратора или фабричной функции	45, 46

В табл. 14.2 перечислены параметры компилятора TypeScript, используемые в данной главе.

Таблица 14.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
module	Задает формат используемых модулей, как описано в главе 5
outDir	Задает каталог, в который будут помещены файлы JavaScript
rootDir	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

14.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

Чтобы приступить к работе с материалом из этой главы, откройте консоль, перейдите в удобное место и создайте папку с именем `decorators`. Выполните команды, показанные в листинге 14.1, чтобы перейти в новую папку и указать менеджеру пакетов Node (NPM) сформировать файл `package.json`, который будет отслеживать пакеты, добавляемые в проект.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 14.1. Создание файла package.json

```
cd decorators
npm init --yes
```

Выполните команды из листинга 14.2 в папке `decorators` для загрузки и установки необходимых пакетов.

Листинг 14.2. Добавление пакетов

```
npm install --save-dev typescript@5.0.2
npm install --save-dev tsc-watch@6.0.0
```

Чтобы создать конфигурационный файл для компилятора TypeScript, добавьте файл с именем `tsconfig.json` в папку `decorators` с содержимым, показанным в листинге 14.3.

Листинг 14.3. Содержимое файла `tsconfig.json` из папки `decorators`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "Node16"
  }
}
```

Эти настройки конфигурации указывают компилятору TypeScript генерировать код для самых последних реализаций JavaScript, используя папку `src` для поиска файлов TypeScript и папку `dist` для выходных данных. Параметр `module` предписывает компилятору использовать тот же механизм, как и в `Node.js` для определения формата модуля.

Чтобы настроить NPM на запуск компилятора и указать формат модуля, добавьте в файл `package.json` конфигурационную запись, как в листинге 14.4.

Листинг 14.4. Настройка NPM в файле `package.json` из папки `decorators`

```
{
  "name": "decorators",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onSuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^6.0.0",
    "typescript": "^5.0.2"
  },
  "type": "module"
}
```

Создайте директорию `decorators/src` и поместите в нее файл с именем `product.ts`, содержимое которого показано в листинге 14.5.

Листинг 14.5. Содержимое файла `product.ts` из папки `src`

```
export class Product {

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

  getDetails(): string {
    return `Name: ${this.name}, Price: ${this.price}`;
  }
}
```

Добавьте в папку `src` файл с именем `city.ts` с содержимым, показанным в листинге 14.6.

Листинг 14.6. Содержимое файла `city.ts` из папки `src`

```
export class City {

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

    getSummary(): string {
        return `Name: ${this.name}, Population: ${this.population}`;
    }
}
```

Чтобы создать точку входа для проекта примера, поместите файл с именем `index.ts` в каталог `src`, содержимое которого показано в листинге 14.7.

Листинг 14.7. Содержимое файла `index.ts` из папки `src`

```
import { City } from "./city.js";
import { Product } from "./product.js";

let city = new City("London", 8_982_000);
let product = new Product("Kayak", 275);

console.log(city.getSummary());
console.log(product.getDetails());
```

Выполните команду из листинга 14.8 в папке `decorators` для запуска компилятора, чтобы скомпилированный код выполнялся автоматически.

Листинг 14.8. Запуск компилятора

```
npm start
```

Компилятор запустится и выдаст следующий результат:

```
08:12:02 - Starting compilation in watch mode...
08:12:04 - Found 0 errors. Watching for file changes.
Name: London, Population: 8982000
Name: Kayak, Price: $275
```

14.2. ЧТО ТАКОЕ ДЕКОРАТОРЫ

Существует несколько типов декораторов. Все они работают практически одинаково, но каждый тип отвечает за преобразование различных аспектов класса:

- *декораторы класса* преобразуют весь класс;
- *декораторы методов* преобразуют метод;
- *декораторы полей* преобразуют поле класса;
- *декораторы аксессоров* преобразуют аксессор класса или автоаксессор.

Декораторы получили свое название от способа их применения, поскольку они используются для «украшения/декорирования» свойств класса, без каких-либо

иных изменений. В листинге 14.9 применен декоратор к классу `Product`. Поскольку декоратор, к которому мы обращаемся, еще не существует, TypeScript выдаст ошибку для этого кода.

Листинг 14.9. Декорирование метода в файле `product.ts` из папки `src`

```
import { time } from "./methodDecorator.js";

export class Product {

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

    @time
    getDetails(): string {
        return `Name: ${this.name}, Price: ${this.price}`;
    }
}
```

Декораторы пишутся с помощью стандартных функций TypeScript/JavaScript и импортируются как любой другой модуль. В примере выше была импортирована функция с именем `time` из файла `methodDecorator.js`. Синтаксис применения декоратора унаследован по сравнению с другими функциями языка: обозначается символом `@`, за которым следует имя декоратора, — в нашей ситуации это `time` (рис. 14.1). Это пример *декоратора метода*, поскольку он был применен к методу.

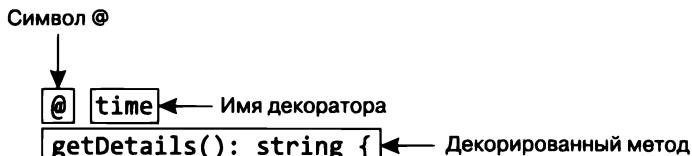


Рис. 14.1. Декоратор метода

Декораторы работают путем замены функции, к которой они применяются. В случае с декоратором метода это означает, что вместо изначального метода средой выполнения JavaScript будет использоваться метод-заменитель. Задачей декоратора `time` из листинга 14.9 является замена метода `getDetails`.

Чтобы определить декоратор, добавьте в папку `src` файл с именем `methodDecorator.ts` со следующим содержимым (листинг 14.10).

Листинг 14.10. Содержимое файла `methodDecorator.ts` из папки `src`

```
export function time(...args) {
    return function() : string {
        return "Hello, Decorator!";
    }
}
```

Декораторы — это функции. Декоратор из листинга 14.10 имеет имя `time` и экспортируется как любая другая функция TypeScript или JavaScript, чтобы его можно было использовать в других файлах кода.

При применении декоратора к методу будет вызвана функция `time`, а возвращаемая ею функция будет использована в качестве замены. В листинге 14.10 есть две функции: внешняя функция-декоратор, определяющая декоратор, и внутренняя функция, которая будет использована в качестве метода-заменителя, как показано на рис. 14.2.

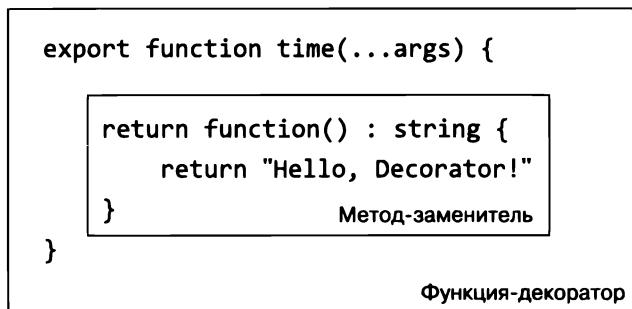


Рис. 14.2. Базовая структура декоратора метода

Сохраните изменения, и после компиляции и выполнения кода вы увидите следующий результат:

```
Name: London, Population: 8982000
Hello, Decorator!
```

Декораторы можно применять к нескольким классам, и каждый из них получит свой метод замены из функции декоратора. В листинге 14.11 к классу `City` применяется тестовый декоратор.

Листинг 14.11. Применение декоратора в файле `city.ts` из папки `src`

```

import { time } from "./methodDecorator.js";

export class City {

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

    @time
    getSummary(): string {
        return `Name: ${this.name}, Population: ${this.population}`;
    }
}

```

Декоратор `time` был применен к двум методам. Функция `time` вызывается один раз для каждого декорированного метода, а возвращаемые замены будут использованы вместо метода, определенного классами, что приведет к следующим результатам:

```
Hello, Decorator!
Hello, Decorator!
```

Заменяющие функции класса, создаваемые декораторами, должны быть подходящими заменами. Метод-заменитель, создаваемый декоратором `time`, не принимает аргументов и возвращает строковое значение, а это означает, что он соответствует сигнатуре декорированных методов. При несовпадении типов, используемых декорированным методом и методом-заменителем, компилятор выдаст ошибку. Листинг 14.12 добавляет в класс `Product` новый метод, возвращающий числовое значение.

Листинг 14.12. Добавление метода в файл `product.ts` из папки `src`

```
import { time } from "./methodDecorator.js";

export class Product {

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

    @time
    getDetails(): string {
        return `Name: ${this.name}, Price: ${this.price}`;
    }

    @time
    getPrice(): number {
        return this.price;
    }
}
```

При таком изменении компилятор сообщит об ошибке:

```
src/product.ts(12,6): error TS1270: Decorator function return type '() => string' is not assignable to type 'void | (() => number)'.
Type '() => string' is not assignable to type '() => number'.
Type 'string' is not assignable to type 'number'.
```

Компилятор TypeScript понимает, что декоратор не дает подходящей замены методу `getPrice`.

14.2.1. Использование контекстных данных декоратора

Декоратор `time` демонстрирует базовую функциональность, но заменяет каждый метод функцией, которая всегда выполняет одно и то же действие, что в реальном проекте бесполезно.

При вызове функции-декоратора метода ей передаются два аргумента, которые мы проигнорировали в предыдущем разделе. Первый аргумент — это исходный метод, к которому был применен декоратор, что позволяет методу-замениителю вызывать исходный метод. Второй аргумент — объект, реализующий интерфейс `ClassMethodDecoratorContext`, который предоставляет полезный

контекст о методе, к которому был применен декоратор. Наиболее полезные члены `ClassMethodDecoratorContext` описаны в табл. 14.3.

Таблица 14.3. Полезные члены `ClassMethodDecoratorContext`

Название	Описание
<code>kind</code>	Это свойство возвращает строку <code>method</code> , указывая, что декоратор был применен к методу. Объекты контекста, предоставляемые для других типов декораторов, определяют это свойство, но возвращают другие значения, как будет показано в последующих примерах
<code>name</code>	Это свойство возвращает значение типа <code>string symbol</code> , содержащее имя метода, к которому применен декоратор
<code>static</code>	Это логическое свойство возвращает <code>true</code> , если декоратор был применен к статическому методу, иначе <code>false</code>
<code>private</code>	Это логическое свойство возвращает <code>true</code> , если декоратор был применен к приватному методу, иначе <code>false</code>
<code>addInitializer()</code>	Этот метод используется для регистрации функции инициализации, как описано в разделе 14.6

Данные аргументы позволяют декоратору использовать функционал исходного метода, как показано в листинге 14.13. (Все утверждения в этом листинге изменились, поэтому я не выделяю их полужирным шрифтом.)

Листинг 14.13. Использование контекста декоратора в файле `methodDecorator.ts` из папки `src`

```
export function time(method: any, ctx: ClassMethodDecoratorContext) {
  const methodName = String(ctx.name);
  return function(this: any, ...args: any[]) {
    const start = performance.now();
    console.log(`${methodName} started`);
    const result = method.call(this, ...args);
    const duration = (performance.now() - start).toFixed(2);
    console.log(`${methodName} ended ${duration} ms`);
    return result;
  }
}
```

Здесь много деталей, поэтому пройдемся по ключевым операторам и посмотрим, что они делают, начиная с объявления функции-декоратора:

```
...
export function time(method: any, ctx: ClassMethodDecoratorContext) {
  ...
}
```

Использование аннотаций типов для декораторов может быть сложным, как показано в следующем примере, и часто бывает проще прибегнуть к типу `any` при написании декоратора, чтобы его можно было применить ко всем методам, как в нашем примере.

Следующий оператор создает постоянное значение типа `string`, содержащее имя метода:

```
...
const methodName = String(ctx.name);
...
```

Свойство `name`, определяемое `ClassMethodDecoratorContext`, возвращает значение `string | symbol`, с учетом того, что имена методов могут быть определены с использованием типа `symbol`, что часто делается в автоматически генерируемом коде для обеспечения уникальности имени метода. Поскольку нам нужно строковое значение, мы возьмем значение `String` и присвоим результат константе. Поместим это утверждение за пределы метода замены, чтобы преобразование в строку выполнялось только один раз для каждого метода, к которому был применен декоратор.

Далее определяется метод замены, который будет вызываться вместо исходного:

```
...
return function(this: any, ...args: any[]) {
...
}
```

Параметры позволяют нам вызывать исходный метод, передавая все полученные аргументы и сохраняя контекст с помощью значения `this` в заменяющем методе:

```
...
const result = method.call(this, ...args);
...
```

В главе 3 была описана функция `method.call`. Ее эффект заключается в том, что декоратор можно применить к любому методу, и заменяющий метод вызовет исходный метод, используя любые полученные аргументы и контекст. Результат присваивается константе, которая используется как результат заменяющего метода, что обеспечивает совместимость с исходным методом.

Остальные операторы в методе замены записывают сообщения, указывающие на момент вызова метода замены и время вызова оригинального метода. Они используют `performance API JavaScript`, который предоставляет таймер с высоким разрешением. Сохраните изменения, и после компиляции и выполнения кода вы увидите вывод, аналогичный приведенному ниже:

```
getSummary started
getSummary ended 4.03 ms
Name: London, Population: 8982000
getDetails started
getDetails ended 0.15 ms
Name: Kayak, Price: $275
```

Выходные данные показывают, что первым вызывается метод `getSummary`, который занял 4,03 миллисекунды, а затем `getDetails` с 0,15 миллисекунды.

В нашем примере декоратор `time` был применен к трем методам, но вывод осуществляется только для двух, поскольку при выполнении кода вызываются

только `getSummary` и `getDetails`. Декоратор также был применен к методу `getPrice`, определенному в листинге 14.12, но этот метод никогда не вызывается и не генерирует выходных данных.

ПРИМЕЧАНИЕ

Разница во времени выполнения возникает из-за того, что в расчеты включается некоторая начальная настройка. В нашем случае, где речь идет об определении декораторов, приводимые цифры не имеют значения. Однако, как правило, измерять производительность следует только после завершения выполнения задач и проведения повторных измерений. Доработанный декоратор, в котором время на инициализацию отделено от времени выполнения, можно найти в разделе 14.7.

14.2.2. Использование определенных типов в декораторе

В листинге 14.13 тип `any` был использован довольно широко, потому что он позволяет легко писать декораторы, которые можно применять к любому методу, независимо от класса, в котором он определен, типов параметров и возвращаемого значения. Такой прием хорошо подходит для декораторов, которые должны вызывать исходный метод, гарантируя при этом сохранение типов.

Для декораторов, которым необходимо знать методы, к которым они применяются, требуется иной подход. В листинге 14.14 определен декоратор с параметрами обобщенного типа.

Листинг 14.14. Добавление параметров типа в файл `methodDecorator.ts` из папки `src`

```
interface HasGetPrice {
    getPrice(): number;
}

export function time<This extends HasGetPrice, Args extends any[], Result>(
    method: (This, Args) => Result,
    ctx: ClassMethodDecoratorContext<This, (This, Args) => Result>)
{
    const methodName = String(ctx.name);
    return function(this: This, ...args: Args) : Result {
        const start = performance.now();
        console.log(`${methodName} started`);
        const result = method.call(this, ...args);
        const duration = (performance.now() - start).toFixed(2);
        console.log(`${methodName} ended ${duration} ms`);
        return result;
    }
}
```

Функция-декоратор имеет три параметра обобщенного типа, представляющих собой тип декорируемого класса, типы аргументов и результат:

```
...
export function time<This extends HasGetPrice, Args extends any[], Result>(
    method: (This, Args) => Result,
    ctx: ClassMethodDecoratorContext<This, (This, Args) => Result>)
{
    ...
}
```

Параметры обобщенного типа влияют на параметры функции декоратора, поэтому параметр `method` является функцией, аннотированной с помощью обобщенных типов:

```
...
export function time<This extends HasGetPrice, Args, Result>(
    method: (This, Args) => Result,
    ctx: ClassMethodDecoratorContext<This, (This, Args) => Result>) {
...
}
```

Параметр `ClassMethodDecoratorContext` имеет два параметра обобщенного типа, которые задают декорируемый класс и сигнатуру декорируемого метода:

```
...
export function test<This extends HasGetPrice, Args, Result>(
    method: (This, Args) => Result,
    ctx: ClassMethodDecoratorContext<This, (This, Args) => Result>) {
...
}
```

Последнее изменение заключается в применении одних и тех же типов к методу замены, гарантуя, что оригинальный метод и его замена используют одни и те же типы:

```
...
return function(this: This, ...args: Args) : Result {
...
}
```

В данном примере тип `This` ограничен таким образом, чтобы декоратор можно было применить только к классам, имеющим метод `getPrice`, который возвращает число `number`, определенное с помощью интерфейса. Только один из двух классов, к которым был применен декоратор, соответствует интерфейсу, поэтому компилятор выдает следующую ошибку:

```
src/city.ts(7,6): error TS1241: Unable to resolve signature of method
decorator when called as an expression.
```

Компилятор использует обобщенные типы декоратора и определяет, что класс `City` не соответствует ограничению на обобщенные типы.

Декораторы методов обычно не ограничивают классы, к которым они могут быть применены, и чаще всего определяют ограничения, специфичные для сигнатуры метода, как показано в листинге 14.15.

Листинг 14.15. Ограничение типа результата в файле `methodDecorator.ts` из папки `src`

```
// interface HasGetPrice {
//     getPrice(): number;
// }

export function time<This, Args extends any[],
    Result extends string | number>(
    method: (This, Args) => Result,
    ctx: ClassMethodDecoratorContext<This, (This, Args) => Result>) {
const methodName = String(ctx.name);
return function(this: This, ...args: Args) : Result {
    const start = performance.now();
    console.log(`${methodName} started');
```

```

const result = method.call(this, ...args);
const duration = (performance.now() - start).toFixed(2);
console.log(`\${methodName} ended \${duration} ms`);
return result;
}
}

```

Параметры обобщенного типа позволяют применять декоратор к любому методу, возвращающему строку или число, без ограничений на другие характеристики класса. После сохранения изменений вы увидите результаты, аналогичные приведенным ниже:

```

getSummary started
getSummary ended 3.90 ms
Name: London, Population: 8982000
getDetails started
getDetails ended 0.12 ms
Name: Kayak, Price: $275

```

14.3. ДРУГИЕ ТИПЫ ДЕКОРАТОРОВ

Методы — это только одна из характеристик класса, для которых можно создавать декораторы. Все типы декораторов работают практически одинаково, но у каждого из них есть свой тип контекстного объекта, как показано в табл. 14.4.

Таблица 14.4. Типы декораторов и контекстные интерфейсы

Характеристика класса	Тип контекста
Класс	ClassDecoratorContext
Методы	ClassMethodDecoratorContext
Поля	ClassFieldDecoratorContext
Аксессоры	ClassGetterDecoratorContext, ClassSetterDecoratorContext
Автоаксессоры	ClassAccessorDecoratorContext

Существует также интерфейс `DecoratorContext`, который представляет собой объединение всех типов контекстов и может использоваться в декораторах, применяемых к различным функциям класса.

В последующих разделах мы создадим каждый тип декоратора и посмотрим, как его можно использовать для преобразования характеристик класса.

14.3.1. Декоратор класса

Конструкторы классов применяются ко всему классу, и наиболее часто этот тип декораторов используется для преобразования путем создания подкласса с новыми функциями. Для декораторов классов требуются параметры обобщенного типа, чтобы предоставить компилятору TypeScript достаточно информации для

предотвращения ошибок. Добавьте в папку `src` файл с именем `classDecorator.ts` с содержимым, показанным в листинге 14.16.

Листинг 14.16. Содержимое файла `classDecorator.ts` из папки `src`

```
export function serialize<T extends new (...args: any) => any>(
    originalClass: T, ctx: ClassDecoratorContext) {

    const className = String(ctx.name);

    return class extends originalClass {

        serialize() {
            console.log(`"${className}": ${JSON.stringify(this)}`);
        }
    };
}
```

Важной частью параметра обобщенного типа является ограничение, без которого компилятор TypeScript выдает ошибку, поскольку тип заменяющего класса не совпадает с исходным. Декоратор добавляет метод `serialize`, который записывает имя класса и JSON-представление объекта, к которому был применен метод.

Имя метода получается из параметра `ClassDecoratorContext`, который определяет свойства и метод, приведенные в табл. 14.5.

Таблица 14.5. Свойства и метод `ClassDecoratorContext`

Название	Описание
<code>kind</code>	Это свойство возвращает строку <code>class</code> , указывающую, что декоратор был применен к классу
<code>name</code>	Это свойство возвращает значение типа <code>string symbol</code> , содержащее имя класса, к которому был применен декоратор
<code>addInitializer()</code>	Этот метод используется для регистрации функции инициализации, как это описано в разделе 14.6

В листинге 14.17 декоратор применен к классу `Product`.

Листинг 14.17. Применение декоратора в файле `product.ts` из папки `src`

```
import { time } from "./methodDecorator.js";
import { serialize } from "./classDecorator.js";

@serialize
export class Product {

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

    @time
    getDetails(): string {
        return `Name: ${this.name}, Price: ${this.price}`;
    }

    @time
    getPrice(): number {
        ...
    }
}
```

```

        return this.price;
    }
}

```

Одним из недостатков декораторов классов является то, что они не изменяют определение преобразуемого типа. Это означает, что метод `serialize`, добавленный декоратором в листинге 14.16, не становится частью типа `Product`, как показано в листинге 14.18.

Листинг 14.18. Вызов дополнительного метода в файле `index.ts` из папки `src`

```

import { City } from "./city.js";
import { Product } from "./product.js";

let city = new City("London", 8_982_000);
let product = new Product("Kayak", 275);

console.log(city.getSummary());
console.log(product.getDetails());

(product as any).serialize();

```

Нам придется прибегнуть к типу `any` для вызова метода, добавленного декоратором, что привело к следующему результату, включающему JSON-представление объекта `Product`:

```

getSummary started
getSummary ended 4.43 ms
Name: London, Population: 8982000
getDetails started
getDetails ended 0.16 ms
Name: Kayak, Price: $275
Product: {"name": "Kayak", "price": 275}

```

Один из способов усовершенствовать типы декораторов — добавить интерфейс и предикатную функцию для защиты типов, как показано в листинге 14.19.

Листинг 14.19. Добавление защиты типа в файл `classDecorator.ts` из папки `src`

```

export function serialize<T extends new (...args: any) => any>(
    originalClass: T, ctx: ClassDecoratorContext) {

    const className = String(ctx.name);

    return class extends originalClass implements Serializable {

        serialize() {
            console.log(`${className}: ${JSON.stringify(this)}`);
        }
    };
}

export interface Serializable {
    serialize();
}

export function isSerializable(target): target is Serializable {
    return typeof target.serialize === "function";
}

```

Интерфейс `Serializeable` и его проверка типов обеспечивают типобезопасный доступ к методу `serialize` для объектов, созданных на основе класса, преобразованного декоратором, как показано в листинге 14.20.

Листинг 14.20. Использование защиты типа в файле index.ts из папки src

```
import { City } from "./city.js";
import { Product } from "./product.js";
import { isSerializeable } from "./classDecorator.js";

let city = new City("London", 8_982_000);
let product = new Product("Kayak", 275);

console.log(city.getSummary());
console.log(product.getDetails());

if (isSerializeable(product)) {
    product.serialize();
}
```

Этот код выдает тот же результат, что и предыдущий пример, но не требует использования типа `any`.

14.3.2. Декораторы полей

Декораторы полей могут изменять начальное значение свойства класса. Добавьте класс с именем `fieldDecorator.ts` в папку `src` с содержимым, показанным в листинге 14.21.

Листинг 14.21. Содержимое файла fieldDecorator.ts из папки src

```
export function double(notused: any, ctx: ClassFieldDecoratorContext) {
    return function(initialValue) {
        return initialValue * 2;
    }
}
```

Декораторы полей возвращают функцию, которая принимает начальное значение поля, а возвращает преобразованное значение. Декоратор `double` умножает начальное значение на 2 и возвращает результат.

Для согласованности с другими типами декораторов функции-декораторы полей определяют два параметра, но первый из них не используется. Второй параметр – это объект контекста, реализующий интерфейс `ClassFieldDecoratorContext`, наиболее полезные возможности которого описаны в табл. 14.6.

Таблица 14.6. Полезные члены ClassFieldDecoratorContext

Название	Описание
<code>kind</code>	Это свойство возвращает строку <code>field</code> , указывающую, что декоратор был применен к полю
<code>name</code>	Это свойство возвращает значение типа <code>string symbol</code> , содержащее имя поля, к которому был применен декоратор

Название	Описание
static	Это логическое свойство возвращает <code>true</code> , если декоратор был применен к статическому полю, иначе — <code>false</code>
private	Это логическое свойство возвращает <code>true</code> , если декоратор был применен к приватному полю, иначе — <code>false</code>
<code>addInitializer()</code>	Этот метод используется для регистрации функции инициализации, как это описано в разделе 14.6

Код из листинга 14.22 добавляет поле в класс `Product` и применяет декоратор `double`.

Листинг 14.22. Добавление декорированного поля в файл `product.ts` из папки `src`

```
import { time } from "./methodDecorator.js";
import { serialize } from "./classDecorator.js";
import { double } from "./fieldDecorator.js";

@serialize
export class Product {
    @double
    private taxRate: number = 20;

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

    @time
    getDetails(): string {
        return `Name: ${this.name}, Price: $$${this.getPrice()}`;
    }

    @time
    getPrice(): number {
        return this.price * (1 + (this.taxRate/100));
    }
}
```

Добавим свойство `taxRate`, которое используется в методе `getPrice` для расчета цены товара на основе значения свойства `price`. Начальное значение, присвоенное полю, равно `20`, но декоратор `double` изменит это значение, что видно из вывода:

```
getSummary started
getSummary ended 4.03 ms
Name: London, Population: 8982000
getDetails started
getPrice started
getPrice ended 0.15 ms
getDetails ended 0.44 ms
Name: Kayak, Price: $385
Product: {"name":"Kayak","price":275,"taxRate":40}
```

Учтите, что декораторы полей преобразуют класс, то есть любое значение, присвоенное конструктором при создании объекта, заменит значение, заданное декоратором.

В листинге 14.23 декоратор поля переделан с целью введения параметров обобщенного типа.

Листинг 14.23. Использование параметров обобщенного типа в файле fieldDecorator.ts из папки src

```
export function double<This, FieldType extends number>(
    notused: any, ctx: ClassFieldDecoratorContext<This, FieldType>) {
    return function (initialValue: FieldType) {
        return initialValue * 2;
    }
}
```

Обобщенные параметры `This` и `FieldType` позволяют накладывать ограничения как на классы, так и на поля, к которым может быть применен декоратор. В листинге 14.23 декоратор ограничен таким образом, чтобы его можно было применять только к числовым полям. Декоратор из листинга 14.23 выдает тот же результат, что и декоратор из листинга 14.22.

14.3.3. Декораторы аксессоров

Декораторы аксессоров (или декораторы доступа) похожи на декораторы методов, поскольку геттеры и сеттеры являются функциями. Добавьте в папку `src` файл с именем `accessorDecorator.ts`, содержимое которого показано в листинге 14.24.

Листинг 14.24. Содержимое файла accessorDecorator.ts из папки src

```
export function log(accessor: any,
    ctx: ClassSetterDecoratorContext | ClassGetterDecoratorContext) {
    const name = String(ctx.name);
    return function(this: any, ...args: any[]) {
        if (ctx.kind === "getter") {
            const result = accessor.call(this, ...args);
            console.log(`${name} get returned ${result}`);
            return result;
        } else {
            console.log(`${name} set to ${args}`);
            return accessor.call(this, ...args);
        }
    }
}
```

Функции декораторов аксессоров принимают два аргумента: исходную функцию аксессора и объект контекста. Тип контекстного объекта будет `ClassSetterDecoratorContext` при декорировании сеттеров и `ClassGetterDecoratorContext` при декорировании геттеров. Эти два типа контекста похожи, а наиболее полезные члены приведены в табл. 14.7.

Декоратор, определенный в листинге 14.24, использует свойство `kind` для определения, был ли декорирован геттер или сеттер. Заменяющая функция для геттеров вызывает исходную, а затем выводит результат в консоль, а для сеттеров выводит полученные аргументы и передает их исходную функцию сеттера. Листинг 14.25 добавляет геттер и сеттер в класс `Product` и декорирует их.

Таблица 14.7. Полезные члены типа контекста аксессора

Название	Описание
kind	Это свойство возвращает строку <code>getter</code> или <code>setter</code> , указывающую, какая часть аксессора была декорирована
name	Это свойство возвращает значение <code>string symbol</code> , содержащее имя аксессора, к которому был применен декоратор
static	Это логическое свойство возвращает <code>true</code> , если декоратор был применен к статическому аксессору, иначе — <code>false</code>
private	Это логическое свойство возвращает <code>true</code> , если декоратор был применен к приватному аксессору, иначе — <code>false</code>
<code>addInitializer()</code>	Этот метод используется для регистрации функции инициализации, как это описано в разделе 14.6

Листинг 14.25. Использование декоратора аксессора в файле `product.ts` из папки `src`

```

import { time } from "./methodDecorator.js";
import { serialize } from "./classDecorator.js";
import { double } from "./fieldDecorator.js";
import { log } from "./accessorDecorator.js";

@serialize
export class Product {
    @double
    private taxRate: number = 20;

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

    @time
    getDetails(): string {
        return `Name: ${this.name}, Price: ${this.getPrice()}`;
    }

    @time
    getPrice(): number {
        return this.price * (1 + (this.taxRate/100));
    }

    @log
    get tax() { return this.taxRate };

    @log
    set tax(newValue) { this.taxRate = newValue};
}

```

В листинге 14.26 используются новые функции `Product`, поэтому вывод, сгенерированный декоратором, будет произведен.

Листинг 14.26. Использование функции класса в файле `index.ts` из папки `src`

```

import { City } from "./city.js";
import { Product } from "./product.js";
import { isSerializable } from "./classDecorator.js";

```

```

let city = new City("London", 8_982_000);
let product = new Product("Kayak", 275);

console.log(city.getSummary());
console.log(product.getDetails());

console.log('Get Product tax: ${product.tax}');
product.tax = 30;

if (isSerializable(product)) {
    product.serialize();
}

```

Данный код генерирует следующий результат, показывая сообщения, созданные декоратором аксессора при использовании геттера и сеттера:

```

getSummary started
getSummary ended 4.08 ms
Name: London, Population: 8982000
getDetails started
getPrice started
getPrice ended 0.23 ms
getDetails ended 0.72 ms
Name: Kayak, Price: $385
tax get returned 40
Get Product tax: 40
tax set to 30
Product: {"name": "Kayak", "price": 275, "taxRate": 30}

```

В листинге 14.27 декоратор аксессора переделан для введения параметров обобщенного типа.

Листинг 14.27. Использование параметров типа в файле accessorDecorator.ts из папки src

```

export function log<This, ValueType extends number>(
    setter: (ValueType) => void,
    ctx: ClassSetterDecoratorContext<This, ValueType>
        : ((ValueType) => void);
export function log<This, ValueType extends number>(
    getter: () => ValueType,
    ctx: ClassGetterDecoratorContext<This, ValueType> ) : () => ValueType;

export function log(accessor: any, ctx: any) {
    const name = String(ctx.name);
    return function(this: any, ...args: any[]) {
        if (ctx.kind === "getter") {
            const result = accessor.call(this, ...args);
            console.log(`${name} get returned ${result}`);
            return result;
        } else {
            console.log(`${name} set to ${args}`);
            return accessor.call(this, ...args);
        }
    }
}

```

Декоратор можно применять к геттерам и сеттерам, каждому из которых требуется своя комбинация типов. Самый простой способ описать эти типы компилятору TypeScript – использовать перегрузки функций, которые позволяют описать тип аксессора и контекстного объекта.

В рамках перегрузки типов функций в листинге 14.27 типы геттеров и сеттеров ограничены так, чтобы декоратор можно было применять только к числовым аксессорам. Декоратор выдает тот же результат, что и в листинге 14.26.

14.3.4. Декораторы автоаксессоров

Последний тип декоратора применяется к автоаксессорам и объединяет геттер и сеттер в один параметр. Добавьте в папку `src` файл с именем `autoAccessorDecorator.ts` с содержимым, показанным в листинге 14.28.

Листинг 14.28. Содержимое файла `autoAccessorDecorator.ts` из папки `src`

```
export function autolog(
    accessor: any,
    ctx: ClassAccessorDecoratorContext) {
    const name = String(ctx.name);
    return {
        get() {
            const result = accessor.get.call(this);
            console.log('Auto-accessor ${name} get returned ${result}');
            return result;
        },
        set(value) {
            console.log('Auto-accessor ${name} set to ${value}');
            return accessor.set.call(this, value);
        },
        init(value) {
            console.log('Auto-accessor initialized to ${value}');
            return value;
        }
    }
}
```

Первым аргументом, принимаемым декоратором, является объект с функциями `get` и `set`, которые соответствуют геттеру и сеттеру, созданным автоаксессором. Второй аргумент представляет собой объект, реализующий интерфейс `ClassAccessorDecoratorContext`, который предоставляет свойства и метод, описанные в табл. 14.8.

Таблица 14.8. Полезные свойства и метод `ClassAccessorDecoratorContext`

Название	Описание
<code>kind</code>	Это свойство возвращает строку <code>accessor</code> , указывающую на то, что аксессор был декорирован
<code>name</code>	Это свойство возвращает <code>string symbol</code> , содержащее имя аксессора, к которому был применен декоратор

Таблица 14.8 (продолжение)

Название	Описание
static	Это логическое свойство возвращает <code>true</code> , если декоратор был применен к статическому аксессору, иначе — <code>false</code>
private	Это логическое свойство возвращает <code>true</code> , если декоратор был применен к приватному аксессору, иначе — <code>false</code>
<code>addInitializer()</code>	Этот метод используется для регистрации функции инициализации, как это описано в разделе 14.6

Результатом работы декоратора является объект, определяющий свойства `get` и `set` с заменяющими их функциями геттера и сеттера, а также свойство `init`, которое вызывается при инициализации декорированного аксессора и результат которого используется для замены начального значения. Свойства `get`, `set` и `init` необязательные, то есть декоратор способен определить только свойства для тех функций, которые требуется преобразовать.

Декоратор в листинге 14.28 регистрирует вызовы геттера и сеттера, а также выводит сообщение во время инициализации. В листинге 14.29 геттер и сеттер, существующие в классе `Product`, заменены декорированным автоаксессором.

Листинг 14.29. Добавление автоаксессора в файл `product.ts` из папки `src`

```
import { time } from "./methodDecorator.js";
import { serialize } from "./classDecorator.js";
import { double } from "./fieldDecorator.js";
import { log } from "./accessorDecorator.js";
import { autolog } from "./autoAccessorDecorator.js";

@serialize
export class Product {
    // @double
    // private taxRate: number = 20;

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

    @time
    getDetails(): string {
        return `Name: ${this.name}, Price: ${this.getPrice()}`;
    }

    @time
    getPrice(): number {
        return this.price * (1 + (this.tax/100));
    }

    // @log
    // get tax() { return this.taxRate };

    // @log
    // set tax(newValue) { this.taxRate = newValue};
```

```

@autolog
accessor tax: number = 20;
}

```

Сохраните изменения, и в выводе появятся сообщения, выдаваемые новым декоратором, примерно так:

```

Auto-accessor initialized to 20
getSummary started
getSummary ended 0.32 ms
Name: London, Population: 8982000
getDetails started
getPrice started
Auto-accessor tax get returned 20
getPrice ended 0.48 ms
getDetails ended 0.93 ms
Name: Kayak, Price: $330
Auto-accessor tax get returned 20
Get Product tax: 20
Auto-accessor tax set to 30
Product: {"name": "Kayak", "price": 275}

```

TypeScript предоставляет встроенные типы интерфейсов для описания автоаксессоров с параметрами обобщенного типа, как показано в листинге 14.30.

Листинг 14.30. Добавление параметров типа в autoAccessorDecorator.ts fi из папки src

```

export function autolog<This, ValueType extends number>(
    accessor: ClassAccessorDecoratorTarget<This, ValueType>,
    ctx: ClassAccessorDecoratorContext<This, ValueType>)
    : ClassAccessorDecoratorResult<This, ValueType> {
    const name = String(ctx.name);
    return {
        get() {
            const result = accessor.get.call(this);
            console.log('Auto-accessor ${name} get returned ${result}');
            return result;
        },
        set(value) {
            console.log('Auto-accessor ${name} set to ${value}');
            return accessor.set.call(this, value);
        },
        init(value) {
            console.log('Auto-accessor initialized to ${value}');
            return value;
        }
    }
}

```

Интерфейс `ClassAccessorDecoratorTarget` используется для представления исходного аксессора и определяет свойства `get` и `set`, которые возвращают типизированные функции. Интерфейс `ClassAccessorDecoratorResult` представляет результат работы декоратора с необязательными свойствами `get`, `set` и `init`. В листинге 14.30 использованы параметры обобщенного типа, чтобы ограничить декоратор так, чтобы его можно было применять только к автоаксессорам типа `number`. Декоратор выдает тот же результат, что и в листинге 14.29.

14.4. ПЕРЕДАЧА ДОПОЛНИТЕЛЬНОГО АРГУМЕНТА ДЕКОРАТОРУ

Декораторы можно создавать внутри фабричной функции, которая получает дополнительный аргумент конфигурации при применении декоратора. Это позволяет настраивать поведение декораторов при их применении к функциям класса. В листинге 14.31 показано добавление фабричной функции в декоратор метода.

Листинг 14.31. Добавление фабричной функции в файл methodDecorator.ts из папки src

```
export function time(label? : string) {
    return function<This, Args extends any[],
        Result extends string | number>(
        method: (This, Args) => Result,
        ctx: ClassMethodDecoratorContext<This,
        (This, Args) => Result>) {
    const methodName = label ?? String(ctx.name);
    return function(this: This, ...args: Args) : Result {
        const start = performance.now();
        console.log(`${methodName} started`);
        const result = method.call(this, ...args);
        const duration = (performance.now() - start).toFixed(2);
        console.log(`${methodName} ended ${duration} ms`);
        return result;
    }
}
}
```

Фабричная функция определяет необязательный строковый параметр, используемый для переопределения имени метода, к которому был применен декоратор, в сообщениях, выводимых на консоль. Обобщенные типы затрудняют чтение декоратора, поэтому в листинге 14.32 они удалены, что также снимает ограничение на результат метода, к которому применен декоратор.

Листинг 14.32. Удаление параметров типа в файле methodDecorator.ts из папки src

```
export function time(label? : string) {
    return function(method, ctx: ClassMethodDecoratorContext) {
        const methodName = label ?? String(ctx.name);
        return function(this, ...args: any[]) {
            const start = performance.now();
            console.log(`${methodName} started`);
            const result = method.call(this, ...args);
            const duration = (performance.now() - start).toFixed(2);
            console.log(`${methodName} ended ${duration} ms`);
            return result;
        }
    }
}
```

Теперь у нас есть три вложенные функции, как показано на рис. 14.3. Внешняя функция – фабрика – принимает необязательную строку и возвращает исходную

функцию-декоратор. Функция-декоратор получает исходный метод и объект контекста и отвечает за возврат метода замены.

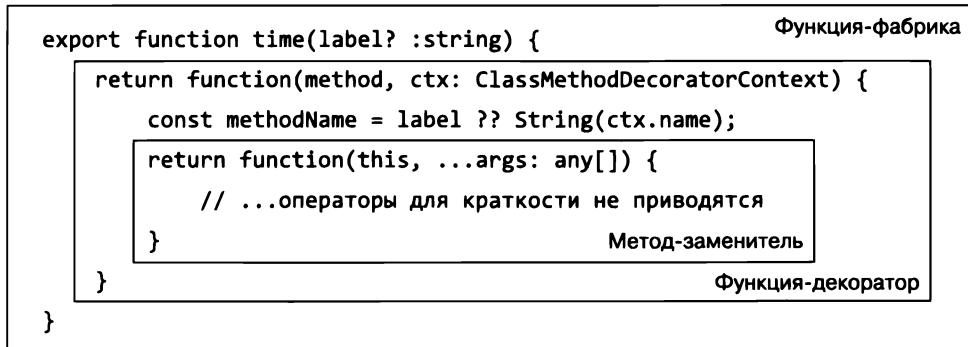


Рис. 14.3. Декоратор с функцией-оберткой

Когда используется фабричная функция, декоратор должен быть применен с круглыми скобками, даже если значение необязательного параметра не указано (листинг 14.33).

Листинг 14.33. Применение обернутого декоратора в файле product.ts из папки src

```

import { time } from "./methodDecorator.js";
import { serialize } from "./classDecorator.js";
import { double } from "./fieldDecorator.js";
import { log } from "./accessorDecorator.js";
import { autolog } from "./autoAccessorDecorator.js";

@serialize
export class Product {

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

    @time("Product.getDetails")
    getDetails(): string {
        return `Name: ${this.name}, Price: ${this.getPrice()}`;
    }

    @time()
    getPrice(): number {
        return this.price * (1 + (this.tax/100));
    }

    @autolog
    accessor tax: number = 20;
}

```

Круглые скобки необходимы везде, где применяется декоратор. Это значит, что класс City также должен быть обновлен, как показано в листинге 14.34.

Листинг 14.34. Применение обернутого декоратора в файле city.ts из папки src

```
import { time } from "./methodDecorator.js";

export class City {

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

    @time()
    getSummary(): string {
        return `Name: ${this.name}, Population: ${this.population}`;
    }
}
```

Сохранив изменения и выполнив код, вы увидите вывод, аналогичный следующему, показывающему, как был использован аргумент, переданный декоратору:

```
Auto-accessor initialized to 20
getSummary started
getSummary ended 0.19 ms
Name: London, Population: 8982000
Product.getDetails started
getPrice started
Auto-accessor tax get returned 20
getPrice ended 0.32 ms
Product.getDetails ended 0.64 ms
Name: Kayak, Price: $330
Auto-accessor tax get returned 20
Get Product tax: 20
Auto-accessor tax set to 30
Product: {"name": "Kayak", "price": 275}
```

Фабричные функции декораторов могут принимать несколько аргументов, но распространенной практикой является принятие объекта, свойства которого используются для настройки декоратора, как показано в листинге 14.35.

Листинг 14.35. Получение объекта конфигурации в файле methodDecorator.ts из папки src

```
type Config = {
    label?: string,
    time?: boolean,
    replacement?: Function,
}

export function time(config? : Config) {
    return function(method, ctx: ClassMethodDecoratorContext) {
        const methodName = config?.label ?? String(ctx.name);
        return function(this, ...args: any[]) {
            const start = performance.now();
            if (config?.time) {
                console.log(`${methodName} started`);
            }
            let result;
            if (config?.replacement) {
                result = config.replacement.call(this, args);
            }
        }
    }
}
```

```
        } else {
            result = method.call(this, args);
        }
        if (config?.time) {
            const duration = (performance.now() - start).toFixed(2);
            console.log(`${methodName} ended ${duration} ms`);
        }
    }
    return result;
}
}
```

Тип **Config** определяет свойства:

- `label`, используемое вместо имени метода;
 - `time`, контролирующее таймер выполнения метода;
 - `replacement`, которое позволяет произвести полную замену исходного метода.

Все свойства `Config` являются необязательными, так что любой параметр конфигурации, который не требуется, может быть опущен.

Листинг 14.36 использует объекты конфигурации при применении декоратора в классе `Product`.

Листинг 14.36. Настройка декоратора в файле product.ts из папки src

```
import { time } from "./methodDecorator.js";
import { serialize } from "./classDecorator.js";
import { double } from "./fieldDecorator.js";
import { log } from "./accessorDecorator.js";
import { autolog } from "./autoAccessorDecorator.js";

@serialize
export class Product {

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

    @time({
        replacement: () => "Hello, Decorator"
    })
    getDetails(): string {
        return `Name: ${this.name}, Price: ${this.getPrice()}`;
    }

    @time({
        label: "Product.getPrice",
        time: true
    })

    getPrice(): number {
        return this.price * (1 + (this.tax/100));
    }

    @autolog
    accessor tax: number = 20;
}
```

В отдельности и в небольшом примере проекта использование объекта для конфигурирования декоратора может показаться избыточным, однако это мощная техника, поскольку она позволяет тонко настраивать поведение декоратора. С этим методом настройки декоратора вы подробнее познакомитесь в части 3, когда мы будем создавать веб-приложения с использованием фреймворка Angular.

В листинге 14.37 добавляется вызов метода `getPrice` в файле `index.ts`, чтобы показать эффект обоих декораторов из листинга 14.36.

Листинг 14.37. Добавление вызова метода в файл `index.ts` из папки `src`

```
import { City } from "./city.js";
import { Product } from "./product.js";
import { isSerializeable } from "./classDecorator.js";

let city = new City("London", 8_982_000);
let product = new Product("Kayak", 275);

console.log(city.getSummary());
console.log(product.getDetails());
console.log('Price: ${product.getPrice()}');

// console.log('Get Product tax: ${product.tax}');
// product.tax = 30;

// if (isSerializeable(product)) {
//   product.serialize();
// }
```

При компиляции и выполнении этого кода будут получены следующие результаты:

```
Auto-accessor initialized to 20
Name: London, Population: 8982000
Hello, Decorator
Product.getPrice started
Auto-accessor tax get returned 20
Product.getPrice ended 0.31 ms
Price: 330
```

14.5. ПРИМЕНЕНИЕ НЕСКОЛЬКИХ ДЕКОРАТОРОВ

К одной функции класса можно применить несколько декораторов, однако необходимо следить, чтобы порядок их выполнения был понятен. Добавьте в папку `src` файл с именем `multiples.ts` с содержимым, показанным в листинге 14.38.

Листинг 14.38. Содержимое файла `multiples.ts` из папки `src`

```
export function message(message: string) {
  console.log(`Factory function: ${message}`);
  return function (method: any, ctx: ClassMemberDecoratorContext) {
    console.log(`Get replacement: ${message}`);
    return function(this: any, ...args: any[]) {
      console.log(`Message: ${message}`);
    }
  }
}
```

```
        return method.call(this, ...args);
    }
}
```

Этот декоратор методов возвращает функцию, которая выводит сообщение перед вызовом исходного метода. В листинге 14.39 декоратор применен к классу `City`.

Листинг 14.39. Применение декоратора в файле city.ts из папки src

```
import { time } from "./methodDecorator.js";
import { message } from "./multiples.js";

export class City {

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

    @message("First Decorator")
    @message("Second Decorator")
    getSummary(): string {
        return `Name: ${this.name}, Population: ${this.population}`;
    }
}
```

В листинге 14.40 код в файле `index.ts` упрощен, чтобы вывод декоратора было легче найти.

Листинг 14.40. Упрощение кода в файле index.ts из папки src

```
import { City } from "./city.js";
import { Product } from "./product.js";
import { isSerializable } from "./classDecorator.js";

let city = new City("London", 8_982_000);
let product = new Product("Kayak", 275);

console.log(city.getSummary());
// console.log(product.getDetails());
// console.log('Price: ${product.getPrice()}');
```

Декораторы выполняются «снаружи внутрь» таким образом, что декоратор, находящийся ближе всего к признаку класса, выполняется последним. Порядок выполнения можно увидеть в выводе, полученном в данном примере:

```
Factory function: First Decorator
Factory function: Second Decorator
Get replacement: Second Decorator
Get replacement: First Decorator
Auto-accessor initialized to 20
Message: First Decorator
Message: Second Decorator

Name: London, Population: 8982000
```

Name: London, Population: 8982000

Для создания метода замены декораторы применяются «изнутри наружу», начиная с самого «внутреннего», так что функция, возвращаемая декоратором,

ближайшим к признаку класса, используется первой, и результат передается следующему декоратору. В листинге 14.41 декоратор изменен таким образом, что метод замены добавляет сообщение к строковому результату.

Листинг 14.41. Использование композиции строк в файле multiples.ts из папки src

```
export function message(message: string) {
    console.log('Factory function: ${message}');
    return function (method: any, ctx: ClassMemberDecoratorContext) {
        console.log('Get replacement: ${message}');
        return function(this: any, ...args: any[]) {
            // console.log('Message: ${message}');
            // return method.call(this, ...args);
            return `${message} (${method.call(this, ...args)})`;
        }
    }
}
```

Сохраните изменения, и вы увидите, как метод замены, созданный внутренним декоратором, передается в качестве входного параметра внешнему декоратору:

```
Factory function: First Decorator
Factory function: Second Decorator
Get replacement: Second Decorator
Get replacement: First Decorator
Auto-accessor initialized to 20

First Decorator (Second Decorator (Name: London, Population: 8982000))
```

14.6. ИНИЦИАЛИЗАТОРЫ

Объекты контекста, предоставляемые декораторам, определяют метод `addInitializer`, который можно использовать для регистрации функции инициализации, как показано в листинге 14.42.

Листинг 14.42. Добавление инициализатора в файл methodDecorator.ts из папки src

```
type Config = {
    label?: string,
    time?: boolean,
    replacement?: Function,
}

export function time(config? : Config) {
    return function(method, ctx: ClassMethodDecoratorContext) {
        let start;
        ctx.addInitializer(() => start = performance.now());
        const methodName = config?.label ?? String(ctx.name);
        return function(this, ...args: any[]) {

            const start = performance.now();
            if (config?.time) {
                console.log(`${methodName} started`);
            }
        }
    }
}
```

```
let result;
if (config?.replacement) {
    result = config.replacement.call(this, args);
} else {
    result = method.call(this, args);
}
if (config?.time) {
    const duration = (performance.now() - start).toFixed(2);
    console.log(`${methodName} ended ${duration} ms`);
}
return result;
}
```

Функция инициализатора передается в метод `addInitializer` объекта контекста и вызывается при создании экземпляра класса, который был декорирован. В нашем примере мы используем инициализатор для вызова метода `performance.now`, что позволяет нам учесть затраты времени на настройку таймера при измерениях, выполненных декоратором. Листинг 14.43 настраивает декоратор для класса `Product`.

Листинг 14.43. Настройка декоратора в файле product.ts из папки src

```
import { time } from "./methodDecorator.js";
import { serialize } from "./classDecorator.js";
import { double } from "./fieldDecorator.js";
import { log } from "./accessorDecorator.js";
import { autolog } from "./autoAccessorDecorator.js";

@serialize
export class Product {

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

    @time({
        //replacement: () => "Hello, Decorator"
        time: true
    })
    getDetails(): string {
        return `Name: ${this.name}, Price: ${this.getPrice()}`;
    }

    @time({
        label: "Product.getPrice",
        time: true
    })
    getPrice(): number {
        return this.price * (1 + (this.tax/100));
    }
    @autolog
    accessor tax: number = 20;
}
```

И наконец, листинг 14.44 изменяет код в файле `index.ts` таким образом, чтобы методы, декорированные в листинге 14.43, были вызваны.

Листинг 14.44. Вызов декорированных методов в файле index.ts из папки src

```
import { City } from "./city.js";
import { Product } from "./product.js";
import { isSerializeable } from "./classDecorator.js";

//let city = new City("London", 8_982_000);
let product = new Product("Kayak", 275);

//console.log(city.getSummary());
console.log(product.getDetails());

console.log('Price: ${product.getPrice()}');
```

Вывод показывает эффект инициализатора: теперь время, затраченное на инициализацию таймера, отделено от измерения отдельных методов:

```
Auto-accessor initialized to 20
getDetails started
Product.getPrice started
Auto-accessor tax get returned 20
Product.getPrice ended 0.32 ms
getDetails ended 0.65 ms
Name: Kayak, Price: $330
Product.getPrice started
Auto-accessor tax get returned 20
Product.getPrice ended 0.40 ms
Price: 330
```

14.7. НАКОПЛЕНИЕ ДАННЫХ О СОСТОЯНИИ

Декораторы могут накапливать данные, что полезно, когда необходимо объединить воздействие декораторов на несколько функций, как показано в листинге 14.45.

Листинг 14.45. Накопление состояния в файле methodDecorator.ts из папки src

```
type Config = {
    label?: string,
    time?: boolean,
    replacement?: Function,
}

const timings = new Map<string, { count: number, elapsed : number}>();

export function writeTimes() {
    [...timings.entries()].forEach(t => {
        const average = (t[1].elapsed / t[1].count).toFixed(2);
        console.log(`${t[0]}, count: ${t[1].count}, time: ${average}ms`);
    });
}

export function time(config? : Config) {
    return function(method, ctx: ClassMethodDecoratorContext) {
        let start;
        ctx.addInitializer(() => start = performance.now());
        const methodName = config?.label ?? String(ctx.name);
```

```

        return function(this, ...args: any[]) {
            start = performance.now();
            // if (config?.time) {
            //     console.log(`${methodName} started`);
            // }
            let result;
            if (config?.replacement) {
                result = config.replacement.call(this, args);
            } else {
                result = method.call(this, args);
            }
            if (config?.time) {
                //const duration = (performance.now() - start).toFixed(2);
                const duration = (performance.now() - start);
                //console.log(`${methodName} ended ${duration} ms`);
                if (timings.has(methodName)) {
                    const data = timings.get(methodName);
                    data.count++;
                    data.elapsed += duration;
                } else {
                    timings.set(methodName, {
                        count: 1, elapsed: duration
                    });
                }
            }
            return result;
        }
    }
}

```

Декоратор использует ассоциативный массив `Map` для отслеживания временных данных для каждого метода, к которому он применяется. Любой метод замены, созданный декоратором, добавляет свои данные в `Map`, накапливая их при каждом вызове метода замены. `Map` определяется вне фабричной функции, декоратора и метода замены, в результате чего используется одна структура `Map` для всех данных.

Данные записываются вызовом функции `writeTimes`, которая экспортится, чтобы ее можно было использовать в других местах приложения, как показано в листинге 14.46.

Листинг 14.46. Запись данных в файл `index.ts` из папки `src`

```

import { City } from "./city.js";
import { Product } from "./product.js";
import { isSerializable } from "./classDecorator.js";
import { writeTimes } from "./methodDecorator.js";

//let city = new City("London", 8_982_000);
let product = new Product("Kayak", 275);

//console.log(city.getSummary());
console.log(product.getDetails());
console.log('Price: ${product.getPrice()}');

writeTimes();

```

В этом примере выводятся данные, аналогичные приведенным ниже, демонстрируя накопление данных:

```
Auto-accessor initialized to 20
Auto-accessor tax get returned 20
Name: Kayak, Price: $330
Auto-accessor tax get returned 20
Price: 330
Product.getPrice, count: 2, time: 0.15ms
getDetails, count: 1, time: 0.24ms
```

Два метода замены, созданные декоратором, были вызваны в общей сложности три раза.

РЕЗЮМЕ

В этой главе вы узнали, как декораторы применяются к классам для преобразования определяемых ими функций. Декораторы не получили широкого распространения за пределами Angular, но ситуация может измениться, когда они будут добавлены в спецификацию языка JavaScript.

- Декораторы – это предлагаемое дополнение к языку JavaScript, позволяющее преобразовывать классы.
- Декораторы применяются с использованием символа @, за которым следует имя декоратора.
- Декораторы можно применять к классам, методам, свойствам, аксессорам и автоаксессорам.
- Декораторы – это функции, которые вызываются с помощью объекта контекста и создают замену функции, к которой они были применены.
- Декораторы могут быть определены с помощью фабричной функции, поддерживающей дополнительные параметры конфигурации.
- Декораторы способны накапливать данные о состоянии, что позволяет нескольким экземплярам декоратора – или нескольким типам декораторов – работать совместно.

В следующей главе вы познакомитесь с тем, как TypeScript работает с кодом JavaScript, когда он является непосредственной частью проекта и когда он находится в сторонних пакетах, от которых зависит приложение.

15

Работа с JavaScript

В этой главе

- ✓ Добавление чистого JavaScript-кода в проект TypeScript.
- ✓ Предоставление определений типов для JavaScript-кода.
- ✓ Включение проверки типов для JavaScript-кода.
- ✓ Определение типов для сторонних пакетов.
- ✓ Использование общедоступных определений типов.
- ✓ Генерация деклараций типов для использования в других проектах.

Проекты TypeScript, как правило, содержат некоторое количество чистого JavaScript-кода, либо потому, что приложение написано на TypeScript и JavaScript, либо потому, что проект зависит от сторонних пакетов JavaScript, установленных с помощью NPM. В данной главе описываются возможности, которые предоставляет TypeScript для работы с JavaScript. В табл. 15.1 приведено краткое содержание главы.

В качестве краткой справки в табл. 15.2 перечислены параметры компилятора TypeScript, используемые в данной главе.

Таблица 15.1. Краткое содержание главы

Задача	Решение	Листинг
Интегрировать файлы JavaScript в проект	Включите параметры компилятора <code>allowJs</code> и <code>checkJs</code>	9–13
Управлять проверкой файла JavaScript компилятором TypeScript	Используйте комментарии <code>@ts-check</code> и <code>@ts-nocheck</code>	14
Описать типы JavaScript	Используйте комментарии JSDoc или создайте файл декларации	15–22
Описать сторонний JavaScript-код	Обновите конфигурации компилятора и создайте файл декларации	22–26
Описать сторонний код, не создавая файл декларации	Используйте пакет, содержащий файл декларации, или установите общедоступную декларацию типов	27–34
Сгенерировать файлы деклараций для использования в других проектах	Включите в компиляторе опцию декларирования	35–38

Таблица 15.2. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
<code>allowJs</code>	Включает в процесс компиляции файлы JavaScript
<code>baseUrl</code>	Задает корневое расположение для разрешения зависимостей модуля
<code>checkJs</code>	Указывает компилятору провести проверку JavaScript-кода на наличие типичных ошибок
<code>declaration</code>	При включении этого параметра компилятор создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода
<code>outDir</code>	Задает каталог, в который будут помещены файлы JavaScript
<code>paths</code>	Задает путь для разрешения зависимостей модуля
<code>rootDir</code>	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
<code>target</code>	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

15.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

Чтобы приступить к работе с материалом из этой главы, откройте новое окно командной строки, перейдите в удобное место и создайте папку с именем `usingjs`. Выполните команды, показанные в листинге 15.1, чтобы перейти в новую папку и указать менеджеру пакетов Node (NPM) сформировать файл `package.json`, который будет отслеживать пакеты, добавляемые в проект.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 15.1. Создание файла package.json

```
cd usingjs  
npm init --yes
```

Выполните команды из листинга 15.2 в папке `usingjs`, чтобы загрузить и установить необходимые пакеты.

Листинг 15.2. Добавление пакетов

```
npm install --save-dev typescript@5.0.2  
npm install --save-dev tsc-watch@6.0.0
```

Чтобы создать конфигурационный файл для компилятора TypeScript, добавьте файл с именем `tsconfig.json` в папку `usingjs` с содержимым, показанным в листинге 15.3.

Листинг 15.3. Содержимое файла tsconfig.json из папки usingjs

```
{  
  "compilerOptions": {  
    "target": "ES2022",  
    "outDir": "./dist",  
    "rootDir": "./src",  
    "module": "Node16"  
  }  
}
```

Эти параметры конфигурации указывают компилятору TypeScript, что он должен генерировать код для самых последних реализаций JavaScript, используя папку `src` для поиска файлов TypeScript и папку `dist` для своих выходных файлов. Параметр `module` указывает компилятору на выбор формата модуля на основе содержимого файла `package.json`.

Чтобы настроить NPM на запуск компилятора и указать формат модуля, добавьте в файл `package.json` конфигурационные записи, показанные в листинге 15.4.

Листинг 15.4. Настройка NPM в файле package.json из папки usingjs

```
{  
  "name": "usingjs",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "tsc-watch --onSuccess \"node dist/index.js\""  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "tsc-watch": "^6.0.0",  
    "typescript": "^5.0.2"  
  },  
  "type": "module"  
}
```

15.1.1. Добавление кода TypeScript в проект примера

Создайте папку `usingjs/src` и поместите в нее файл `product.ts` с кодом, показанным в листинге 15.5.

Листинг 15.5. Содержимое файла `product.ts` из папки `src`

```
export class Product {

    constructor(public id: number,
                public name: string,
                public price: number) {
        // Не требуется никаких операторов
    }
}

export enum SPORT {
    Running, Soccer, Watersports, Other
}

export class SportsProduct extends Product {
    private _sports: SPORT[];

    constructor(public id: number,
                public name: string,
                public price: number,
                ...sportArray: SPORT[]) {
        super(id, name, price);
        this._sports = sportArray;
    }

    usedForSport(s: SPORT): boolean {
        return this._sports.includes(s);
    }

    get sports(): SPORT[] {
        return this._sports;
    }
}
```

В этом файле определен базовый класс `Product`, который расширяется классом `SportsProduct`, добавляющим функции, характерные для спортивных товаров.

Далее добавьте в папку `src` файл `cart.ts` с кодом, показанным в листинге 15.6.

Листинг 15.6. Содержимое файла `cart.ts` из папки `src`

```
import { SportsProduct } from "./product.js";

class CartItem {

    constructor(public product: SportsProduct, public quantity: number) {
        // Не требуется никаких операторов
    }

    get totalPrice(): number {
        return this.quantity * this.product.price;
    }
}
```

```

export class Cart {
    private items = new Map<number, CartItem>();

    constructor(public customerName: string) {
        // не требуется никаких операторов
    }

    addProduct(product: SportsProduct, quantity: number): number {
        if (this.items.has(product.id)) {
            let item = this.items.get(product.id);
            item.quantity += quantity;
            return item.quantity;
        } else {
            this.items.set(product.id, new CartItem(product, quantity));
            return quantity;
        }
    }

    get totalPrice(): number {
        return [...this.items.values()].reduce((total, item) =>
            total += item.totalPrice, 0);
    }

    get itemCount(): number {
        return [...this.items.values()].reduce((total, item) =>
            total += item.quantity, 0);
    }
}

```

Здесь определяется класс `Cart`, который отслеживает выбор покупателем объектов `SportProduct` с помощью `Map`.

Чтобы создать точку входа в проект, поместите в папку `src` файл под названием `index.ts` с кодом, показанным в листинге 15.7.

Листинг 15.7. Содержимое файла `index.ts` из папки `src`

```

import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
    SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);

let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

console.log('Cart has ${cart.itemCount} items');
console.log('Cart value is $$({cart.totalPrice.toFixed(2)})');

```

Код в файле `index.ts` создает несколько объектов `SportsProduct`, использует их для заполнения корзины `Cart` и записывает данные о ее содержимом в консоль.

Выполните команду из листинга 15.8 в папке `usingjs`, чтобы запустить компилятор и автоматически выполнить скомпилированный код.

Листинг 15.8. Запуск компилятора

```
npm start
```

Компилятор выдаст следующий результат:

```
7:23:34 AM - Starting compilation in watch mode...7:23:36 AM - Found
0 errors. Watching for file changes.
Cart has 4 items
Cart value is $341.30
```

15.2. РАБОТА С JAVASCRIPT

Все примеры, приведенные в книге, предполагают, что вы работаете исключительно на TypeScript. Зачастую это невозможно либо потому, что TypeScript вводится в проект частично, либо потому, что вам необходимо работать с JavaScript-кодом, который уже был разработан в предыдущих проектах.

Проект может содержать как TypeScript, так и JavaScript-код параллельно, требуя лишь изменений в компиляторе TypeScript и некоторых дополнительных шагов по описанию типов, используемых в коде JavaScript. Для демонстрации этого процесса приведу некоторый JavaScript-код. Добавьте в папку `src` файл `formatters.js` с кодом из листинга 15.9.

ПРИМЕЧАНИЕ

Расширение файла в листинге 15.9 — `.js`, поскольку в нем чистый JavaScript-код. Важно использовать правильное расширение для примеров в данном разделе.

Листинг 15.9. Содержимое файла `formatters.js` из папки `src`

```
export function sizeFormatter(thing, count) {
    writeMessage('The ${thing} has ${count} items');
}

export function costFormatter(thing, cost) {
    writeMessage('The ${thing} costs $$ ${cost.toFixed(2)}', true);
}

function writeMessage(message) {
    console.log(message);
}
```

Этот JavaScript-файл экспортирует две функции форматирования, которые выводят сообщения в консоль. Чтобы внедрить код JavaScript в приложение, добавьте в файл `index.ts` утверждения, показанные в листинге 15.10.

Листинг 15.10. Использование функций JavaScript в файле `index.ts` из папки `src`

```
import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";
import { sizeFormatter, costFormatter } from "./formatters.js";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
    SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);
```

```
let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", cart.totalPrice);
```

После сохранения изменений в файле `index.ts` компилятор запустится без проблем, однако при выполнении кода отобразится следующее сообщение:

```
Error [ERR_MODULE_NOT_FOUND]: Cannot find module 'formatters.js' imported
from index.js
```

Компилятор TypeScript без труда находит JavaScript-код, но не копирует его в папку `dist`, то есть среда выполнения Node.js не может обнаружить JavaScript-код во время выполнения.

15.2.1. Включение JavaScript в процесс компиляции

Компилятор TypeScript использует файлы JavaScript для разрешения зависимостей во время компиляции, но не включает их в генерируемый вывод. Чтобы изменить это поведение, установите опцию `allowJs` в файле `tsconfig.json` в значение `true`, как показано в листинге 15.11.

Листинг 15.11. Изменение конфигурации в файле `tsconfig.json` из папки `usingjs`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "Node16",
    "allowJs": true
  }
}
```

Данная опция включает в процесс компиляции JavaScript-файлы из папки `src`. Файлы JavaScript не содержат функций TypeScript, однако компилятор преобразует их в соответствии с версией JavaScript, указанной в настройке `target`, и форматом модуля, указанным в свойстве `module`.

В нашем примере никакие функции кода, используемые в файле `formatters.js`, не изменятся, поскольку свойство `target` установлено в `ES2022`, а свойство `module` указывает компилятору читать формат модуля из файла `package.json`. Однако настройка компилятора TypeScript на включение файлов JavaScript позволяет легко смешивать код и обеспечивает согласованность версий функций JavaScript.

15.2.2. Проверка типа JavaScript-кода

Компилятор TypeScript проверит JavaScript-код на наличие типичных ошибок, когда параметр конфигурации `checkJs` установлен в значение `true` (листинг 15.12). Подобная проверка не такая углубленная, как проверка файлов TypeScript, но она позволяет выявить потенциальные проблемы.

Листинг 15.12. Настройка компилятора в файле tsconfig.json из папки usingjs

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "Node16",
    "allowJs": true,
    "checkJs": true
  }
}
```

Компилятор не всегда обнаруживает изменение свойства `checkJs` до перезапуска. Сохранив файл `tsconfig.json`, остановите компилятор с помощью комбинации клавиш `Control+C`. Для его повторного запуска выполните команду, показанную в листинге 15.13, в папке `usingjs`.

Листинг 15.13. Запуск компилятора

```
npm start
```

Функция `costFormatter` в файле `formatters.js` вызывает функцию `writeMessage`, определенную в том же файле, с большим количеством аргументов, чем имеется параметров. В JavaScript это допустимо, так как в этом языке программирования нет ограничений на количество аргументов при вызове функции. Однако компилятор TypeScript сообщит об ошибке, которая считается довольно распространенной.

```
src/formatters.js(6,60): error TS2554: Expected 0-1 arguments, but got 2.
```

Эта функция полезна только в том случае, если вы можете модифицировать файлы JavaScript, чтобы устраниТЬ проблемы, о которых сообщает компилятор. Возможно, у вас есть код, из-за которого компилятор TypeScript выдаст ошибку, но который нельзя изменить, поскольку он соответствует требованиям сторонней библиотеки. Если у вас есть несколько редактируемых и нередактируемых JavaScript-файлов, вы можете добавить комментарии, чтобы контролировать, какие из них проверяются. В табл. 15.3 описаны комментарии, которые применяются к верхней части файлов JavaScript.

Таблица 15.3. Комментарии, контролирующие проверку JavaScript

Название	Описание
<code>//@ts-check</code>	Включает проверку содержимого JavaScript-файла, даже если свойство <code>checkJs</code> в файле <code>tsconfig.json</code> равно <code>false</code>
<code>//@ts-nocheck</code>	Отключает проверку содержимого JavaScript-файла, даже если свойство <code>checkJs</code> в файле <code>tsconfig.json</code> равно <code>true</code>

В листинге 15.14 в файл `formatters.js` добавлен комментарий, запрещающий компилятору проверять содержимое текущего файла. Однако это не влияет на проверку других JavaScript-файлов в проекте, если только к ним не будет применен аналогичный комментарий.

Листинг 15.14. Отключение проверок JavaScript в файле formatters.js из папки src

```
// @ts-nocheck

export function sizeFormatter(thing, count) {
    writeMessage(`The ${thing} has ${count} items`);
}

export function costFormatter(thing, cost) {
    writeMessage(`The ${thing} costs $$ ${cost.toFixed(2)}`, true);
}

function writeMessage(message) {
    console.log(message);
}
```

Компилятор обнаружит это изменение и выполнит его без проверки утверждений в файле JavaScript, выдав следующий результат:

```
The Cart has 4 items
The Cart costs $341.30
```

15.3. ОПИСАНИЕ ТИПОВ, ИСПОЛЬЗУЕМЫХ В КОДЕ JAVASCRIPT

Компилятор TypeScript включит JavaScript-код в проект, но статическая информация о типах в нем будет недоступна. Компилятор предпримет попытку самостоятельно вывести используемые типы в JavaScript-коде. Однако это может быть затруднительно, и он прибегнет к использованию типа `any`, особенно для параметров функций и их результатов. Например, функция `costFormatter`, определенная в файле `formatters.js`, будет восприниматься, как если бы она была определена с помощью этих аннотаций типов:

```
...
export function costFormatter(thing: any, cost: any): any {
...
}
```

Добавление JavaScript в проект, вероятно, создаст дыры в проверке типов, которые сведут на нет преимущества использования TypeScript. Компилятор не сможет определить, что функция `costFormatter` предполагает получение числового значения, в чем можно убедиться, добавив в файл `index.ts` оператор, предоставляющий строковое значение, как показано в листинге 15.15.

Листинг 15.15. Использование неправильного типа в файле index.ts из папки src

```
import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";
import { sizeFormatter, costFormatter } from "./formatters.js";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
    SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);
```

```
let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", '${cart.totalPrice}');
```

Новое утверждение вызывает функцию `costFormatter` с двумя строковыми аргументами. Компилятор TypeScript скомпилирует код без ошибок, но при его выполнении функция `costFormatter` вызывает метод `toFixed`, не проверив, что ей передали числовое значение. Это приводит к следующей ошибке:

```
writeMessage('The ${thing} costs $$${cost.toFixed(2)}', true);  
^
```

`TypeError: cost.toFixed is not a function`

Данную проблему можно решить, предоставив компилятору информацию о типе, описывающую JavaScript-код, чтобы его использование можно было проверить в процессе компиляции. Существует два подхода к описанию типов в коде JavaScript, с которыми вы познакомитесь в следующих разделах.

15.3.1. Использование комментариев для описания типов

Компилятор TypeScript может получать информацию о типе, если она включена в комментарии JSDoc. JSDoc — это популярный язык разметки, используемый для аннотирования JavaScript-кода в виде комментариев. Листинг 15.16 добавляет JSDoc-комментарии в файл `formatters.js`.

СОВЕТ

Многие редакторы кода помогают генерировать комментарии JSDoc. Например, Visual Studio Code реагирует на создание комментария и автоматически генерирует список параметров функции.

Листинг 15.16. Использование JSDoc в файле `formatters.js` из папки `src`

```
// @ts-nocheck

export function sizeFormatter(thing, count) {
    writeMessage('The ${thing} has ${count} items');
}

/**
 * Format something that has a money value
 * @param { string } thing - the name of the item
 * @param { number } cost - the value associated with the item
 */
export function costFormatter(thing, cost) {
    writeMessage('The ${thing} costs $$${cost.toFixed(2)}', true);
}

function writeMessage(message) {
    console.log(message);
}
```

Спецификация JSDoc позволяет указывать типы для параметров функции. Комментарий JSDoc в листинге 15.16 указывает, что функция `costFormatter` ожидает получить параметры типа `string` и `number`. Информация о типе является стандартной частью JSDoc, но обычно она служит для справки.

Компилятор TypeScript анализирует комментарии JSDoc, чтобы получить информацию о типе JavaScript-кода. При сохранении JSDoc-комментария в листинге 15.16 компилятор запустится и сообщит следующее:

```
Argument of type 'string' is not assignable to parameter of type 'number'.
```

Компилятор прочитал комментарий JSDoc к функции `costFormatter` и определил, что значение, используемое для вызова функции в файле `index.ts`, имеет не тот тип данных.

СОВЕТ

Полный список тегов JSDoc, которые понимает компилятор TypeScript, приведен по ссылке <https://github.com/Microsoft/TypeScript/wiki/JSDoc-support-in-JavaScript>.

В комментариях JSDoc может использоваться синтаксис TypeScript для описания более сложных типов, как показано в листинге 15.17, где используется объединение типов.

Листинг 15.17. Описание объединения типов в файле `formatters.js` из папки `src`

```
// @ts-nocheck

export function sizeFormatter(thing, count) {
    writeMessage('The ${thing} has ${count} items');
}

/**
 * Format something that has a money value
 * @param { string } thing - the name of the item
 * @param { number | string } cost - the value associated with the item
 */
export function costFormatter(thing, cost) {
    if (typeof cost === "number") {
        writeMessage('The ${thing} costs $$${cost.toFixed(2)}', true);
    } else {
        writeMessage('The ${thing} costs $$${cost}');
    }
}

function writeMessage(message) {
    console.log(message);
}
```

Функция `costFormatter` была модифицирована таким образом, чтобы она могла принимать числовые и строковые значения для своего параметра `cost`, что отражено в обновленном комментарии JSDoc, в котором тип указан как `number | string`. После сохранения изменений код будет скомпилирован, и на выходе получится следующий результат:

```
The Cart has 4 items
The Cart costs $341.3
```

15.3.2. Файлы деклараций типов

Файлы деклараций, называемые также *файлами определения типов*, предоставляют способ описания JavaScript-кода в файле TypeScript без изменения файла исходного кода. Файлы деклараций имеют расширение `d.ts`, а имя файла соответствует файлу JavaScript. Чтобы сформировать файл декларации для файла `formatters.js`, необходимо создать файл с именем `formatters.d.ts`. Добавьте в папку `src` файл с именем `formatters.d.ts`, содержимое которого показано в листинге 15.18.

Листинг 15.18. Содержимое файла `formatters.d.ts` из папки `src`

```
export declare function sizeFormatter(thing: string, count: number): void;
export declare function
    costFormatter(thing: string, cost: number | string ): void;
```

Содержимое файла декларации типов соответствует содержимому описываемого им файла с кодом. Каждое утверждение содержит ключевое слово `declare`, которое сообщает компилятору, что данное утверждение описывает типы, определенные в другом месте. В листинге 15.18 описаны параметры и типы результатов функций, экспортруемых из файла `formatters.js`.

СОВЕТ

Файлы декларации типов имеют приоритет над комментариями JSDoc, если они оба используются для описания JavaScript-кода.

Файл декларации типов должен описывать все функции, определенные в соответствующем JavaScript-файле, потому что этот JavaScript-файл является единственным источником информации, используемым компилятором TypeScript, который, в свою очередь, больше не обращается к исходному JavaScript-файлу. В нашем примере это означает, что декларация типа в листинге 15.18 должна описывать функции `sizeFormatter` и `costFormatter`, так как они используются в файле `index.ts`. Если какая-то функция не описана в файле декларации типов, то она не будет видна компилятору TypeScript. В листинге 15.19 продемонстрировано, как можно изменить функцию `writeMessage` в файле `formatters.js`, чтобы сделать ее доступной для экспорта и использования в других частях приложения.

СОВЕТ

Компилятор TypeScript полагается на точность содержимого файла определения типов, а это означает, что вы несете ответственность за то, чтобы выбранные вами типы соответствовали поддерживаемым кодом JavaScript, и что все функции в коде JavaScript реализованы так, как вы описали.

Листинг 15.19. Экспорт функции в файл `formatters.js` из папки `src`

```
// @ts-nocheck

export function sizeFormatter(thing, count) {
    writeMessage('The ${thing} has ${count} items');
}
```

```
/**
 * Format something that has a money value
 * @param { string } thing - the name of the item
 * @param { number | string } cost - the value associated with the item
 */
export function costFormatter(thing, cost) {
  if (typeof cost === "number") {
    writeMessage(`The ${thing} costs $$${cost.toFixed(2)}`, true);
  } else {
    writeMessage(`The ${thing} costs $$${cost}`);
  }
}

export function writeMessage(message) {
  console.log(message);
}
```

В листинге 15.20 используется новая экспортированная функция в файле `index.ts` для вывода простого сообщения.

Листинг 15.20. Использование функции в файле `index.ts` из папки `src`

```
import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";
import { sizeFormatter, costFormatter, writeMessage }
  from "./formatters.js";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
  SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);

let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);
writeMessage("Test message");
```

Компилятор обработает изменения в файле `index.ts` при его сохранении и выдаст следующую ошибку:

```
Module '/usingjs/src/formatters' has no exported member 'writeMessage'.
```

Для описания содержимого модуля `formatters` компилятор полностью полагается на файл декларации типов. Для того чтобы функция `writeMessage` стала видимой для компилятора, необходимо добавить в файл `formatters.d.ts` соответствующую декларацию, как показано в листинге 15.21.

Листинг 15.21. Добавление утверждения в файл `formatters.d.ts` из папки `src`

```
export declare function sizeFormatter(thing: string, count: number): void;
export declare function
  costFormatter(thing: string, cost: number | string ): void;
export declare function writeMessage(message: string): void;
```

После того как функция будет включена в файл определения, код в проекте скомпилируется и выведет следующий результат:

```
The Cart has 4 items
The Cart costs $341.3
Test message
```

15.3.3. Описание стороннего JavaScript-кода

Файлы деклараций также допускается использовать для описания JavaScript-кода, добавленного в проект из сторонних пакетов, установленных с помощью NPM. Откройте новое окно командной строки, перейдите в папку `usingjs` и выполните команду, показанную в листинге 15.22, чтобы установить новый пакет.

Листинг 15.22. Добавление пакета в проект примера

```
npm install debug@4.3.4
```

Пакет `debug` — это пакет утилит, который предоставляет оформленный вывод отладки в консоль JavaScript. Данный пакет небольшой, но хорошо написан и широко используется в разработке на JavaScript.

Компилятор попытается вывести типы для сторонних пакетов, но с тем же ограниченным успехом, что и для файлов JavaScript в проекте. Для пакетов, установленных в папке `node_modules`, можно создать файл декларации типов, но это несколько неудобно; лучше использовать общедоступные определения (подробнее об этом рассказывается в следующем разделе).

Первый шаг — перенастроить способ, которым компилятор TypeScript разрешает зависимости от модулей, как показано в листинге 15.23.

Листинг 15.23. Настройка компилятора в файле `tsconfig.json` из папки `usingjs`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "Node16",
    "allowJs": true,
    "checkJs": true,
    "baseUrl": ".",
    "paths": {
      "*": ["types/*.d.cts", "types/*.d.mts", "types/*.d.ts"]
    }
  }
}
```

Свойство `paths` используется для указания мест, которые компилятор TypeScript будет использовать при попытке разрешить операторы `import` для модулей. Конфигурация, используемая в листинге, предписывает компилятору искать все пакеты в папке `types`. Я указал расширения файлов `cts`, `mts` и `ts`, что очень важно, поскольку декларации типов должны соответствовать форматам модулей пакета,

к которому они применяются. (Как уже объяснялось в главе 5, расширение `mjs` обозначает файл TypeScript, использующий модули ECMAScript, а `cjs` – модули CommonJS. Файлы с этими расширениями компилируются в файлы с расширениями `mjs` и `cjs` и представляют собой альтернативу файлу `package.json` для указания формата модуля.)

При использовании свойства `paths` также необходимо указать свойство `baseUrl`, а значение, используемое в листинге, сообщит компилятору, что местоположение, указанное свойством `path`, можно найти в той же папке, что и файл `tsconfig.json`.

Следующим шагом будет создание папки `usingjs/types` и добавление в нее файла с именем `debug.d.ts`. Здесь используется расширение `cts`, поскольку пакет отладки публикуется в виде модулей CommonJS (это я выяснил, изучив файл `package.json` и JavaScript-файлы в репозитории проекта на GitHub).

После создания файла добавьте в него содержимое, показанное в листинге 15.24.

Листинг 15.24. Содержимое файла `debug.d.cts` из папки `types`

```
declare interface Debug {
    (namespace: string): Debugger
}
declare interface Debugger {
    (...args: string[]): void;
    enabled: boolean;
}

declare var debug: Debug & { default: Debug };

export = debug;
```

Процесс описания стороннего модуля может быть сложным прежде всего потому, что авторы пакета могли не предполагать, что кто-то попытается описать их код с помощью статических типов. Еще более усложняет ситуацию широкий спектр версий языка JavaScript и форматов модулей: чтобы предоставить TypeScript описания, которые будут полезными и будут точно отражать код в модуле, могут потребоваться различные «танцы с бубном».

Два интерфейса в листинге 15.24 описывают базовые возможности пакета `debug`, что позволяет настроить и использовать простой отладчик. Последние два утверждения необходимы для представления экспорта из пакета в TypeScript.

СОВЕТ

Для получения подробной информации о полном API, предоставляемом пакетом `debug`, посетите страницу <https://github.com/debug-js/debug>.

Чтобы воспользоваться пакетом `debug`, добавьте операторы, показанные в листинге 15.25, в файл `index.ts` в папке `src`.

Листинг 15.25. Использование пакета в файле `index.ts` из папки `src`

```
import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";
import { sizeFormatter, costFormatter, writeMessage }
```

```

from "./formatters.js";
import debug from "debug";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
    SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);

let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

let db = debug("Example App", true);
db.enabled = true;
db("Message: %s", "Test message");

```

Компилятор TypeScript обнаружит файл декларации и определит, что функция `debug` была вызвана избыточным количеством аргументов, выдав следующее сообщение об ошибке:

```

...
src/index.ts(20,31): error TS2554: Expected 1 arguments, but got 2.
...

```

Эта ошибка не была бы выявлена без файла декларации, поскольку в чистом JavaScript не требуется, чтобы количество аргументов, используемых для вызова функции, совпадало с количеством параметров, которые она определяет (см. главу 8).

Чтобы убедиться, что компилятор нашел файл декларации, не обязательно создавать преднамеренную ошибку. Вместо этого откройте новое окно командной строки, перейдите в папку `usingjs` и выполните команду, показанную в листинге 15.26.

Листинг 15.26. Запуск компилятора

```
tsc --traceResolution
```

Аргумент `traceResolution`, который также можно использовать в качестве параметра конфигурации в файле `tsconfig.json`, указывает компилятору на необходимость информировать о ходе попыток найти каждый модуль. Вывод может быть весьма подробным, особенно в сложных проектах, но для нашего примера трассировка будет содержать такое сообщение:

```
===== Module name 'debug' was successfully resolved to 'C:/usingjs/types/
debug.d.cts'. =====
```

У вас может быть указано другое местоположение, но это сообщение подтверждает, что компилятор нашел файл пользовательской декларации и будет использовать его для разрешения зависимостей от пакета `debug`.

НЕ СОЗДАВАЙТЕ ДЕКЛАРАЦИИ ДЛЯ СТОРОННИХ ПАКЕТОВ

Файл декларации в листинге 15.24 показывает, что можно описывать общедоступные пакеты, но я не рекомендую этого делать и не привожу никаких подробностей о различных способах описания содержимого пакета.

Во-первых, точное представление чужого кода довольно затруднительно, а создание файла декларации типа может потребовать детального анализа пакета и глубокого понимания того, что он делает и как работает. Во-вторых, пользовательские декларации, как правило, фокусируются только на тех функциях, которые требуются в данный момент, а файлы декларации исправляются и расширяются по мере необходимости дополнительных функций, что приводит к результатам, которыми трудно управлять. В-третьих, каждый новый выпуск означает, что файл декларации должен быть пересмотрен, чтобы можно было убедиться, что он по-прежнему точно отражает API, представленный пакетом.

Но самая веская причина не создавать файлы декларации типов заключается в том, что существует отличная библиотека высококачественных деклараций для тысяч пакетов JavaScript, доступная благодаря проекту Definitely Typed (DT), описанному в следующем разделе. Кроме того, рост популярности TypeScript означает, что все больше пакетов будут поставляться со встроенными файлами деклараций типов.

Если вы настроены писать собственные файлы или внести свой вклад в проект DT, то воспользуйтесь специальным руководством по описанию пакетов от Microsoft, которое можно найти по адресу <https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html>.

15.3.4. Использование файлов деклараций проекта Definitely Typed

Проект Definitely Typed предоставляет файлы деклараций для тысяч пакетов JavaScript и является более надежным и быстрым способом работы со сторонними пакетами, чем создание собственных файлов деклараций. Файлы деклараций DT устанавливаются с помощью команды `npm install`. Чтобы установить файл декларации для пакета `debug`, выполните в папке `usingjs` команду, показанную в листинге 15.27.

Листинг 15.27. Установка пакета объявления типов

```
npm install --save-dev @types/debug
```

Имя, используемое для пакета DT, начинается с `@types/`, за которым следует название пакета, для которого требуется описание. Например, для пакета `debug` пакет DT называется `@types/debug`.

СОВЕТ

Обратите внимание, что в листинге 15.27 не указан номер версии пакета `@types/debug`. При установке пакетов `@types` я предполагаю, чтобы NPM самостоятельно выбирал версию пакета.

Компилятор не будет использовать декларации DT до тех пор, пока не будет изменена конфигурация, запрещающая компилятору искать их в папке `types`, как показано в листинге 15.28.

ПРИМЕЧАНИЕ

Изменение конфигурации необходимо, так как в нашем случае присутствуют как пользовательские, так и DT-объявления для одного и того же пакета. В реальных проектах это не будет проблемой, и вы сможете использовать параметры конфигурации для выбора между пользовательскими и DT-декларациями для каждого используемого пакета.

Листинг 15.28. Настройка компилятора в файле `tsconfig.json` из папки `usingjs`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "Node16",
    "allowJs": true,
    "checkJs": true,
    // "baseUrl": ".",
    // "paths": {
    //   "*": ["types/*.d.cts", "types/*.d.mts", "types/*.d.ts"]
    // },
  }
}
```

Откройте новое окно командной строки, перейдите в папку `usingjs` и выполните команду, показанную в листинге 15.29, чтобы увидеть эффект от использования пакета DT.

Листинг 15.29. Запуск компилятора

```
tsc --traceResolution
```

Новая трассировка показывает, что компилятор нашел другой файл декларации.

```
===== Type reference directive 'debug' was successfully resolved to
'C:/usingjs/node_modules/@types/debug/index.d.ts' with Package ID
```

```
'@types/debug/index.d.ts@4.1.7', primary: true. =====
```

Компилятор просматривает каталог `node_modules/@types`, который содержит папки, соответствующие каждому из пакетов, для которых существуют файлы декларации, следуя тому же шаблону, что и для пользовательских файлов. (Нет необходимости вносить изменения в конфигурацию, чтобы указать компилятору искать в папке `node_modules/@types`.)

В результате используется файл декларации DT, содержащий полное описание API, представленного пакетом `debug`. В листинге 15.30 откорректировано количество аргументов, применяемых для вызова функции `debug`.

Листинг 15.30. Использование возможностей пакета в файле `index.ts` из папки `src`

```
import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";
import { sizeFormatter, costFormatter, writeMessage }
    from "./formatters.js";
import debug from "debug";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
    SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);

let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

let db = debug("Example App");
db.enabled = true;
db("Message: %s", "Test message");
```

Сохраните изменения и запустите компилятор TypeScript с помощью команды `npm start`, если он еще не запущен. Компилятор запустится, используя новый файл декларации, который включает описание метода `debug` из листинга. Скомпилированный код выдает следующий результат:

```
The Cart has 4 items
The Cart costs $341.3
Example App Message: Test message +0ms
```

15.3.5. Использование пакетов, включающих декларации типов

С ростом популярности TypeScript пакеты стали включать файлы деклараций, что исключает необходимость загружать дополнительные файлы. Самый простой способ проверить, содержит ли проект файл декларации, — установить пакет и посмотреть в папку `node_modules`. Для наглядности откройте новое окно командной строки, перейдите в каталог `usingjs` и выполните команду, показанную в листинге 15.31, чтобы добавить пакет в проект примера.

Листинг 15.31. Добавление пакета в проект

```
npm install chalk@4.1.2
```

Пакет Chalk предоставляет стили для консольного вывода. Изучив содержимое папки `node_modules/chalk`, вы увидите, что в ней находится файл `index.d.ts`, содержащий декларации типов для пакета.

Чтобы убедиться, что компилятор TypeScript может найти файл декларации Chalk, добавьте строки, показанные в листинге 15.32, в файл `index.ts` в папке `src`.

Листинг 15.32. Добавление утверждений в файл `index.ts` из папки `src`

```
import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";
import { sizeFormatter, costFormatter, writeMessage }
    from "./formatters.js";
import debug from "debug";
import chalk from "chalk";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
    SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);

let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

console.log(chalk.greenBright("Formatted message"));
console.log(chalk.notAColor("Formatted message"));
```

Одной из функций, предоставляемых пакетом Chalk, является раскраска текста, выводимого в консоль. Первое выражение указывает Chalk применить ярко-зеленый цвет `greenBright`, а второе использует несуществующее свойство. После сохранения изменений в файле `index.ts` компилятор будет использовать файл декларации и выдаст следующую ошибку:

```
src/index.ts(22,19): error TS2339: Property 'notAColor' does not exist on
type 'Chalk & ChalkFunction & { supportsColor: false | ColorSupport; Level:
Level; Color: Color; ForegroundColor: ForegroundColor; BackgroundColor:
BackgroundColor; Modifiers: Modifiers; stderr: Chalk & { ...; }; }'.
```

Чтобы увидеть, как компилятор находит файл декларации, воспользуйтесь командной строкой и запустите команду из листинга 15.33 в папке `usingjs`.

Листинг 15.33. Запуск компилятора

```
tsc --traceResolution
```

Вывод аргумента `traceResolution` содержит много информации, но если вы внимательно изучите сообщения, то увидите, в каких местах компилятор проверяет наличие файлов объявлений и как влияют настройки Chalk в файле `package.json`:

```
===== Module name 'chalk' was successfully resolved to 'C:
(usingjs/node_modules/chalk/index.d.ts' with Package ID
'chalk/index.d.ts@4.1.2'. =====
```

В листинге 15.34 удален оператор, который намеренно вызвал ошибку компилятора, поэтому пример приложения может быть скомпилирован и выполнен. При этом утверждение, отформатированное Chalk, будет отображаться ярко-зеленым цветом.

Листинг 15.34. Удаление утверждения в файле index.ts из папки src

```
import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";
import { sizeFormatter, costFormatter, writeMessage }
    from "./formatters.js";
import debug from "debug";
import chalk from "chalk";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
    SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);

let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

console.log(chalk.greenBright("Formatted message"));

//console.log(chalk.notAColor("Formatted message"));
```

Код будет скомпилирован и выполнен, а оператор, отформатированный Chalk, отобразится ярко-зеленым цветом.

15.4. ГЕНЕРАЦИЯ ФАЙЛОВ ДЕКЛАРАЦИЙ

Если ваш код будет использоваться в других проектах, вы можете с помощью компилятора генерировать файлы деклараций вместе с чистым JavaScript. Это сохранит информацию о типах для других программистов TypeScript, но при этом позволит использовать проект как обычный JavaScript.

Компилятор не будет создавать файлы декларации, когда включена опция `allowJS`, поэтому необходимо удалить зависимость от файла `formatters.js`, чтобы проект полностью состоял из TypeScript. Добавьте в папку `src` файл `tsFormatters.ts` и поместите в него код из листинга 15.35.

Листинг 15.35. Содержимое файла `tsformatters.ts` из папки `src`

```
export function sizeFormatter(thing: string, count: number): void {
    writeMessage(`The ${thing} has ${count} items`);
}

export function costFormatter(thing: string, cost: number | string): void {
    if (typeof cost === "number") {
        writeMessage(`The ${thing} costs $$ ${cost.toFixed(2)})`);
```

```

    } else {
      writeMessage(`The ${thing} costs $$ ${cost}`);
    }
}

export function writeMessage(message: string): void {
  console.log(message);
}

```

Это JavaScript-код из файла `formatters.js`, но с аннотациями типов. В листинге 15.36 файл `index.ts` обновлен таким образом, чтобы зависеть от файла TypeScript, а не от файла JavaScript.

ВНИМАНИЕ

Важно выполнить все изменения в этом процессе, поскольку отключение опции `allowJS` только предотвращает добавление JavaScript-файла в выходную папку. Это не мешает коду TypeScript зависеть от файла JavaScript, что может привести к ошибкам во время выполнения, так как среда выполнения JavaScript не сможет найти все необходимые файлы.

Листинг 15.36. Обновление зависимости в файле `index.ts` из папки `src`

```

import { SportsProduct, SPORT } from "./product.js";
import { Cart } from "./cart.js";
import { sizeFormatter, costFormatter, writeMessage }
  from "./tsFormatters.js";
import debug from "debug";
import chalk from "chalk";

let kayak = new SportsProduct(1, "Kayak", 275, SPORT.Watersports);
let hat = new SportsProduct(2, "Hat", 22.10, SPORT.Running,
  SPORT.Watersports);
let ball = new SportsProduct(3, "Soccer Ball", 19.50, SPORT.Soccer);

let cart = new Cart("Bob");
cart.addProduct(kayak, 1);
cart.addProduct(hat, 1);
cart.addProduct(hat, 2);

sizeFormatter("Cart", cart.itemCount);
costFormatter("Cart", `${cart.totalPrice}`);

console.log(chalk.greenBright("Formatted message"));

//console.log(chalk.notAColor("Formatted message"));

```

В листинге 15.37 изменена конфигурация компилятора для отключения свойств `allowJS` и `checkJS` и включения автоматической генерации файлов деклараций.

Листинг 15.37. Настройка компилятора в файле `tsconfig.json` из папки `usingjs`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "allowJs": true,
    "checkJs": true
  }
}
```

```
"outDir": "./dist",
"rootDir": "./src",
"module": "Node16",
// "allowJs": true,
// "checkJs": true,
"declaration": true
}
}
```

Компилятор безопасно сгенерирует файлы деклараций только после перезапуска. Нажмите **Control+C** для остановки компилятора, а затем выполните в папке `usingjs` команду, показанную в листинге 15.38, чтобы запустить его заново.

Листинг 15.38. Запуск компилятора

```
npm start
```

Если свойство `declaration` имеет значение `true`, компилятор создаст в папке `dist` файлы деклараций, описывающие функции, экспортимые из каждого файла TypeScript, как показано на рис. 15.1.

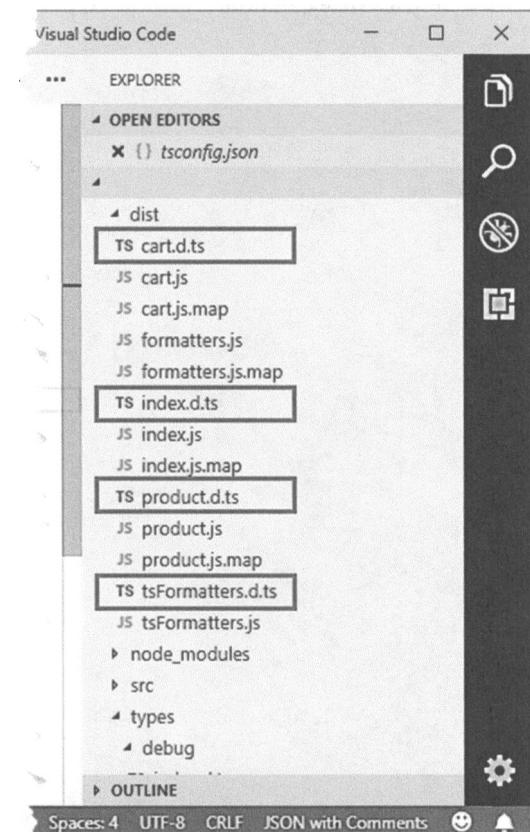


Рис. 15.1. Генерация файлов деклараций

РЕЗЮМЕ

В этой главе вы узнали, как работать с JavaScript в TypeScript-проекте. Мы разобрались с тем, как настроить компилятор для обработки и проверки типов JavaScript-файлов и как использовать файлы деклараций для описания JavaScript-кода компилятору. Проекты TypeScript могут включать в себя чистый JavaScript-код.

- Компилятор TypeScript может проверять типы в чистом JavaScript-коде, но способен вывести только базовые типы без деклараций типов.
- Декларации типов могут быть предоставлены для JavaScript-кода как в проекте, так и в сторонних пакетах.
- Публичные декларации типов доступны для большинства популярных пакетов JavaScript, а многие пакеты включают в себя декларации типов.
- Компилятор TypeScript может генерировать декларации типов для созданного им JavaScript-кода, что позволяет использовать скомпилированный код в других проектах на TypeScript.

В следующей части книги мы создадим ряд веб-приложений, основанных на TypeScript, начиная с автономного приложения, а затем с использованием фреймворков Angular и React.

Часть III

*Создание веб-приложений
на основе TypeScript*

Создание автономного веб-приложения, часть 1

В этой главе

- ✓ Настройка сборщика (или бандлера, от англ. bundler) для создания файлов и эффективной их доставки в браузеры.
- ✓ Настройка компилятора TypeScript для работы с JSX.
- ✓ Использование JSX-файлов для объединения HTML-разметки и кода TypeScript.
- ✓ Запуск простого веб-приложения без использования фреймворка.

В этой части книги вы увидите, как TypeScript вписывается в процесс разработки с наиболее популярными фреймворками для веб-приложений: Angular и React. В обоих случаях мы пройдем весь путь создания проекта: от написания простого веб-приложения до настройки веб-сервиса.

В текущей же главе мы создадим веб-приложение без помощи каких-либо фреймворков. Это позволит нам получить базовые представления о возможностях фреймворков и понять контекст использования функций TypeScript.

Я не рекомендую создавать реальные приложения без применения фреймворка, но работа над автономным приложением раскрывает много аспектов TypeScript и его роль в современной разработке, что важно для обучения. В качестве краткой справки в табл. 16.1 перечислены параметры компилятора TypeScript, используемые в главе.

Таблица 16.1. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
jsx	Определяет, как обрабатываются HTML-элементы в файлах JSX/TSX
jsxFactory	Задает имя фабричной функции для замены HTML-элементов в файлах JSX/TSX
outDir	Задает каталог, в который будут помещены файлы JavaScript
rootDir	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

16.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

Прежде чем приступить к работе, откройте новое окно командной строки, перейдите в удобное место и создайте папку `webapp`. Выполните команды, показанные в листинге 16.1, чтобы переместиться в папку `webapp` и указать менеджеру пакетов Node (NPM) сформировать файл с именем `package.json`.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 16.1. Создание файла package.json

```
cd webapp
npm init --yes
```

Создадим инструментарий, включающий компилятор TypeScript, чтобы показать типичный рабочий процесс разработки веб-приложений. Для этого необходимо установить в проект пакет TypeScript локально; нельзя полагаться на глобально установленный пакет из главы 1. Для установки пакета TypeScript выполните команду из листинга 16.2 в папке `webapp`.

Листинг 16.2. Добавление пакетов с помощью менеджера пакетов node

```
npm install --save-dev typescript@5.0.2
```

Дополнительные пакеты будут установлены по мере развития приложения, но пока достаточно пакета TypeScript. Чтобы настроить компилятор TypeScript, добавьте файл с именем `tsconfig.json` в папку `webapp` с содержимым, показанным в листинге 16.3.

Листинг 16.3. Содержимое файла tsconfig.json в папке webapp

```
{
  "compilerOptions": {
    "target": "es2022",
```

```

    "outDir": "./dist",
    "rootDir": "./src"
}
}

```

В этой конфигурации указано, что компилятор должен использовать JavaScript версии ES2022, искать файлы кода в папке `src`, а также помещать сгенерированные файлы в каталог `dist`. Чтобы начать работу, создайте папку `src` и добавьте в нее файл `index.ts` с содержимым, показанным в листинге 16.4.

Листинг 16.4. Содержимое файла index.ts из папки src

```
console.log("Web App");
```

Запустите команды из листинга 16.5 в папке `webapp` для компиляции `index.ts` и выполнения содержимого полученного JavaScript-файла.

Листинг 16.5. Компиляция и выполнение результата

```
tsc
node dist/index.js
```

Скомпилированный код выдаст следующий результат:

```
Web App
```

16.2. СОЗДАНИЕ ИНСТРУМЕНТАРИЯ

Разработка веб-приложений опирается на цепочку инструментов, которые компилируют код и подготавливают его к доставке и выполнению приложения средой исполнения JavaScript. Компилятор TypeScript на данный момент — единственный инструмент разработки в текущем проекте (рис. 16.1).

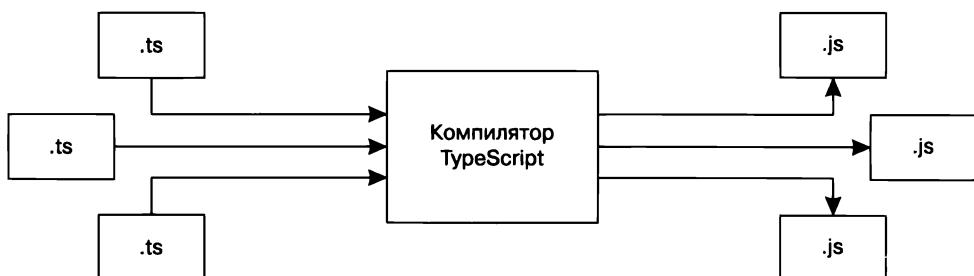


Рис. 16.1. Инструментарий начального проекта

Инструменты разработки скрыты, когда вы используете фреймворк, такой как Angular или React, как будет показано в последующих главах. Однако в этой главе мы установим и настроим каждый инструмент и увидим, как они взаимодействуют.

16.3. ДОБАВЛЕНИЕ СБОРЩИКА

Когда вы запускаете приложение с Node.js в папке проекта, любые операторы `import` могут быть разрешены с помощью генерированного компилятором TypeScript кода JavaScript или пакетов, установленных в папке `node_modules`.

Среда выполнения JavaScript начинает работу с точки входа в приложение — файла `index.js`, скомпилированного из файла `index.ts`, — и обрабатывает содержащиеся в нем операторы `import`. Для каждого оператора `import` среда выполнения разрешает зависимости и загружает требуемый модуль, который будет представлять собой другой JavaScript-файл. Все операторы `import`, объявленные в новом JavaScript-файле, обрабатываются аналогичным образом, что позволяет разрешить все зависимости в приложении и начать выполнение кода.

Среде выполнения JavaScript заранее не известно, какие инструкции `import` содержатся в каждом из файлов кода, и поэтому она не знает, какие файлы JavaScript необходимы. Но это и неважно, поскольку поиск файлов для разрешения зависимостей — относительно быстрая операция, так как все локальные файлы легко доступны.

Такой метод не очень хорошо подходит для веб-приложений, у которых нет прямого доступа к файловой системе. Вместо этого файлы приходится запрашивать по протоколу HTTP, а это довольно медленная и ресурсозатратная операция и не позволяет легко проверить несколько каталогов для разрешения зависимостей между файлами. Вместо этого используется сборщик (бандлер), который разрешает зависимости во время компиляции и упаковывает все файлы, используемые приложением, в один файл. Один HTTP-запрос доставляет весь JavaScript, необходимый для работы приложения, а другие типы контента, например CSS, могут быть включены в файл, создаваемый бандлером, который называется *пакетом*. Код и содержимое во время процесса упаковки могут быть уменьшены в размере и сжаты. Это позволяет сократить объем данных, который нужно передать, и уменьшить пропускную способность канала, необходимую для доставки приложения клиенту. Большие приложения можно разделить на несколько пакетов, чтобы дополнительный код или контент могли загружаться отдельно и только тогда, когда это необходимо.

Наиболее широко используемый сборщик — webpack, и он является ключевой частью инструментария, используемого в React и Angular, хотя обычно не требуется работать с ним напрямую, как вы увидите в последующих главах. Работать с webpack непросто, но он поддерживается огромным количеством пакетов, позволяющих создавать цепочки инструментов разработки практически для любого типа проектов. Выполните команды из листинга 16.6 в папке `webapp`, чтобы добавить пакеты webpack в проект примера.

Листинг 16.6. Добавление пакетов в проект примера

```
npm install --save-dev webpack@5.76.3
npm install --save-dev webpack-cli@5.0.1
npm install --save-dev ts-loader@9.4.2
```

Пакет `webpack` содержит основные функции сборки, а пакет `webpack-cli` добавляет поддержку командной строки. Для работы с различными типами содержимого Webpack использует пакеты, называемые *загрузчиками*, а пакет `ts-loader` добавляет поддержку компиляции файлов TypeScript и передачи скомпилированного кода в созданный `webpack`-пакет.

Для настройки `webpack` добавьте в папку `webapp` файл с именем `webpack.config.js` с содержимым, показанным в листинге 16.7.

Листинг 16.7. Содержимое файла `webpack.config.js` из папки `webapp`

```
module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: "./src/index.ts",
  output: { filename: "bundle.js" },
  resolve: { extensions: [".ts", ".js"] },
  module: {
    rules: [
      { test: /\.ts/, use: "ts-loader", exclude: /node_modules/ }
    ]
  }
};
```

В этих настройках `entry` и `output` указывают `webpack` начинать работу с файла `src/index.ts` при разрешении зависимостей приложения и присваивать файлу пакета имя `bundle.js`. Остальные параметры конфигурируют `webpack` для использования пакета `ts-loader` для обработки файлов с расширением `ts`.

СОВЕТ

Подробности о полном спектре поддерживаемых `webpack` конфигурационных параметров см. на сайте <https://webpack.js.org>.

Выполните команду из листинга 16.8 в папке `webapp`, чтобы запустить `webpack` и создать файл пакета.

Листинг 16.8. Создание файла пакета

```
npx webpack
```

`Webpack` обрабатывает все зависимости в проекте и использует пакет `ts-loader` для компиляции найденных им файлов TypeScript, что приводит к следующему результату:

```
asset bundle.js 788 bytes [emitted] (name: main)
./src/index.ts 25 bytes [built] [code generated]
webpack 5.17.0 compiled successfully in 1865 ms
```

В папке `dist` создается файл `bundle.js`. Запустите команду, показанную в листинге 16.9, в папке `webapp` для выполнения кода в пакете.

Листинг 16.9. Выполнение файла пакета

```
node dist/bundle.js
```

В проекте есть только один файл TypeScript, но пакет является автономным и останется таковым даже при усложнении приложения из примера. Выполнение пакета приводит к следующему результату:

Web App

Добавление webpack и связанных с ним пакетов изменило цепочку инструментов разработки, как показано на рис. 16.2.

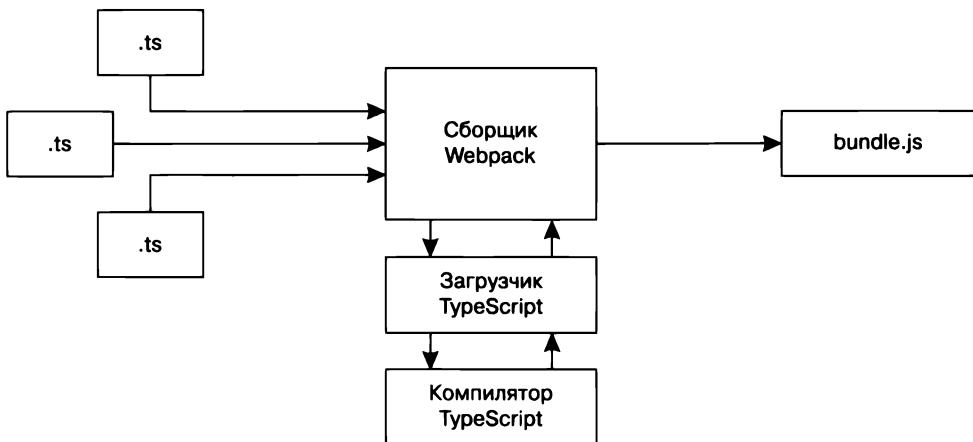


Рис. 16.2. Добавление пакета в инструментарий

16.4. ДОБАЛЕНИЕ ВЕБ-СЕРВЕРА РАЗРАБОТКИ

Для доставки файла пакета в браузер и его выполнения необходим веб-сервер. Webpack Dev Server (WDS) – это HTTP-сервер, интегрированный в webpack, который включает поддержку автоматической перезагрузки браузера при изменении файлов кода и создании нового файла пакета. Запустите команду из листинга 16.10 в папке webapp, чтобы установить WDS.

Листинг 16.10. Добавление пакета wds

```
npm install --save-dev webpack-dev-server@4.13.1
```

Измените конфигурацию webpack, чтобы задать базовые настройки для WDS, как показано в листинге 16.11.

Листинг 16.11. Конфигурация сервера в файле webpack.config.js из папки webapp

```
module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: "./src/index.ts",
  output: { filename: "bundle.js" },
  resolve: { extensions: [".ts", ".js"] },
```

```

module: {
  rules: [
    { test: /\.ts/, use: "ts-loader", exclude: /node_modules/ }
  ]
},
devServer: {
  static: "./assets",
  port: 4500
}
};

```

Новые параметры конфигурации предписывают WDS искать любой файл, не являющийся пакетом, в папке с именем `assets` и прослушивать HTTP-запросы на порте 4500. Чтобы предоставить WDS HTML-файл, который можно использовать для ответа браузерам, создайте папку `webapp/assets` и поместите в нее файл `index.html` с содержимым из листинга 16.12.

Листинг 16.12. Содержимое файла index.html из папки assets

```

<!DOCTYPE html>
<html>
<head>
  <title>Web App</title>
  <script src="bundle.js"></script>
</head>
<body>
  <div id="app">Web App Placeholder</div>
</body>
</html>

```

Когда браузер получает HTML-файл, он обработает его содержимое и встретит элемент `script`, который вызывает HTTP-запрос к файлу `bundle.js`, содержащему JavaScript-код приложения.

Для запуска сервера выполните в папке `webapp` команду, показанную в листинге 16.13.

Листинг 16.13. Запуск веб-сервера разработки

```
pxr webpack serve
```

Запустится HTTP-сервер, и пакет будет создан. Однако папка `dist` больше не используется для хранения файлов. Результат процесса упаковки хранится в памяти и используется для ответа на HTTP-запросы без необходимости создания файла на диске. Как только сервер запустится и приложение будет упаковано, вы увидите следующий результат:

```

<i> [webpack-dev-server] Project is running at:
<i> [webpack-dev-server] Loopback: http://localhost:4500/
<i> [webpack-dev-server] On Your Network (IPv4): http://192.168.1.13:4500/
<i> [webpack-dev-server] Content not from webpack is served from './assets'
directory
asset bundle.js 609 KiB [emitted] (name: main)
runtime modules 27.3 KiB 12 modules
modules by path ./node_modules/ 173 KiB

```

```
modules by path ./node_modules/webpack-dev-server/client/ 68.9 KiB 16
modules
  modules by path ./node_modules/webpack/hot/*.js 4.59 KiB
  ./node_modules/webpack/hot/dev-server.js 1.88 KiB [built] [code generated]
  ./node_modules/webpack/hot/log.js 1.34 KiB [built] [code generated]
    + 2 modules
  modules by path ./node_modules/html-entities/lib/*.js 81.3 KiB
    ./node_modules/html-entities/lib/index.js 7.74 KiB [built]
  [code generated]
    ./node_modules/html-entities/lib/named-references.js 72.7 KiB [built]
  [code generated]
    ./node_modules/html-entities/lib/numeric-unicode-map.js 339 bytes
  [built] [code generated]
    ./node_modules/html-entities/lib/surrogate-pairs.js 537 bytes [built]
  [code generated]
    ./node_modules/ansi-html-community/index.js 4.16 KiB [built] [code
generated]
    ./node_modules/events/events.js 14.5 KiB [built] [code generated]
./src/index.ts 24 bytes [built] [code generated]
```

webpack 5.76.3 compiled successfully in 3883 ms

Детали сообщений не так важны, они просто дают представление о ходе процесса. После запуска сервера откройте веб-браузер и перейдите по адресу `http://localhost:4500`, который является портом, на котором настроен WDS на прием HTTP-запросов. Браузер отобразит содержимое файла `index.html`, как показано на рис. 16.3.

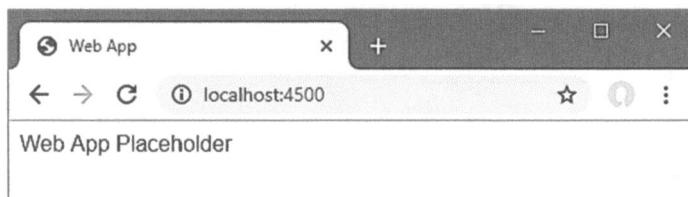


Рис. 16.3. Отображение HTML-файла

Откройте инструменты разработчика в браузере (**F12**) и переключитесь на вкладку `Console` для просмотра вывода `console.log` в файле `index.ts`:

Web App

Когда WDS запущен, webpack переходит в режим наблюдения, в котором при обнаружении изменений в файлах кода происходит сборка нового пакета. В процессе формирования пакета WDS встраивает в JavaScript-файл дополнительный код, который открывает обратное соединение с сервером и ожидает сигнала на перезагрузку браузера, который посыпается для каждого нового пакета. В результате браузер автоматически перезагружается при каждом обнаружении и обработке изменений, что можно увидеть, добавив в файл `index.ts` утверждение, как показано в листинге 16.14.

СОВЕТ

Функция перезагрузки работает только для файлов кода и не распространяется на HTML-файл из папки assets. Изменения в HTML-файле вступают в силу только при перезапуске WDS.

Листинг 16.14. Добавление утверждения в файл index.ts из папки src

```
console.log("Web App");
console.log("This is a new statement");
```

После сохранения файла `index.ts` webpack создаст новый пакет, а браузеру отправится сигнал на перезагрузку, в результате чего в консоли инструментов разработчика браузера (F12) появится следующий вывод:

```
Web App
This is a new statement
```

Добавление WDS расширяет инструментарий и связывает приложение со средой выполнения JavaScript, предоставляемой браузером, как показано на рис. 16.4.



Рис. 16.4. Добавление WDS в набор инструментов для разработки

Эта цепочка инструментов включает ключевые элементы, которые вы найдете в большинстве проектов веб-приложений, хотя отдельные части часто скрыты от глаз. Обратите внимание, что компилятор TypeScript является лишь одним из звеньев этой цепочки, позволяющим интегрировать код TypeScript в набор более широких инструментов разработки JavaScript.

16.5. СОЗДАНИЕ МОДЕЛИ ДАННЫХ

Приложение получает список товаров из веб-сервиса с помощью HTTP-запроса. Пользователь сможет выбирать товары для формирования заказа, который будет отправлен обратно в веб-сервис другим HTTP-запросом. Для запуска модели данных создайте папку `src/data` и поместите в нее файл `entities.ts` с кодом, показанным в листинге 16.15.

Листинг 16.15. Содержимое файла entities.ts из папки src/data

```
export type Product = {
    id: number,
    name: string,
    description: string,
    category: string,
    price: number
};

export class OrderLine {
    constructor(public product: Product, public quantity: number) {
        // не требуется никаких операторов
    }

    get total(): number {
        return this.product.price * this.quantity;
    }
}

export class Order {
    private lines = new Map<number, OrderLine>();

    constructor(initialLines?: OrderLine[]) {
        if (initialLines) {
            initialLines.forEach(ol => this.lines.set(ol.product.id, ol));
        }
    }

    public addProduct(prod: Product, quantity: number) {
        if (this.lines.has(prod.id)) {
            if (quantity === 0) {
                this.removeProduct(prod.id);
            } else {
                this.lines.get(prod.id)!.quantity += quantity;
            }
        } else {
            this.lines.set(prod.id, new OrderLine(prod, quantity));
        }
    }

    public removeProduct(id: number) {
        this.lines.delete(id);
    }

    get orderLines(): OrderLine[] {
        return [...this.lines.values()];
    }

    get productCount(): number {
        return [...this.lines.values()]
            .reduce((total, ol) => total += ol.quantity, 0);
    }

    get total(): number {
        return [...this.lines.values()]
            .reduce((total, ol) => total += ol.total, 0);
    }
}
```

Типы `Product`, `Order` и `OrderLine` экспортуются, чтобы их можно было использовать за пределами файла кода. Класс `Order` представляет выбранные пользователем товары, каждый из которых выражается в виде объекта `OrderLine`, объединяющего товар (`Product`) и `quantity`. Определим `Product` как псевдоним типа. Это упростит работу с данными, полученными удаленно (мы еще столкнемся с этим в главе 17). Типы `Order` и `OrderLine` определены как классы, поскольку они определяют дополнительные функции, выходящие за переделы коллекции связанных свойств.

16.5.1. Создание источника данных

Сперва создадим класс, обеспечивающий доступ к некоторым локальным тестовым данным. Чтобы упростить переход от локальных данных к удаленным, определим абстрактный класс, предоставляющий основные функции, и создадим конкретные реализации для каждого источника данных. Добавьте файл `abstractDataSource.ts` в папку `src/data` и используйте его для определения класса (листинг 16.16).

ВНИМАНИЕ

Обратите внимание, что операторы импорта в этой главе не включают расширения файлов `js`. Требование к расширениям файлов специфично для `Node.js` и их интерпретации спецификации языка `JavaScript`. В текущей главе для выполнения скомпилированного `JavaScript`-кода используется браузер, и это требует опускать расширения файлов. Надеюсь, что подобная несовместимость будет устранена, но я не ожидаю, что этот вопрос решится в ближайшее время.

Листинг 16.16. Содержимое файла `abstractdatasource.ts` из папки `src/data`

```
import { Product, Order } from "./entities";

export type ProductProp = keyof Product;

export abstract class AbstractDataSource {
    private _products: Product[];
    private _categories: Set<string>;
    public order: Order;
    public loading: Promise<void>;

    constructor() {
        this._products = [];
        this._categories = new Set<string>();
        this.order = new Order();
        this.loading = this.getData();
    }

    async getProducts(sortProp: ProductProp = "id",
        category?: string): Promise<Product[]> {
        await this.loading;
        return this.selectProducts(this._products, sortProp, category);
    }
}
```

```

protected async getData(): Promise<void> {
    this._products = [];
    this._categories.clear();
    const rawData = await this.loadProducts();
    rawData.forEach(p => {
        this._products.push(p);
        this._categories.add(p.category);
    });
}

protected selectProducts(prods: Product[],
    sortProp: ProductProp, category?: string): Product[] {
    return prods.filter(p=> category === undefined
        || p.category === category)
        .sort((p1, p2) => p1[sortProp] < p2[sortProp]
            ? -1 : p1[sortProp] > p2[sortProp] ? 1: 0);
}

async getCategories(): Promise<string[]> {
    await this.loading;
    return [...this._categories.values()];
}

protected abstract loadProducts(): Promise<Product[]>;
abstract storeOrder(): Promise<number>;
}

```

Класс `AbstractDataSource` использует возможности JavaScript `Promise` для получения данных в фоновом режиме и применяет ключевые слова `async/await` для выражения кода, зависящего от этих операций. Класс в листинге 16.16 вызывает в конструкторе абстрактный метод `loadProducts`, а методы `getProducts` и `getCategories` ожидают завершения фоновой операции перед возвращением ответов. Для создания реализации класса источника данных, использующего локальные тестовые данные, добавьте в папку `src/data` файл `localDataSource.ts` и поместите в него код из листинга 16.17.

Листинг 16.17. Содержимое файла `localdatasource.ts` из папки `src/data`

```

import { AbstractDataSource } from "./abstractDataSource";
import { Product } from "./entities";

export class LocalDataSource extends AbstractDataSource {

    loadProducts(): Promise<Product[]> {
        return Promise.resolve([
            { id: 1, name: "P1", category: "Watersports",
                description: "P1 (Watersports)", price: 3 },
            { id: 2, name: "P2", category: "Watersports",
                description: "P2 (Watersports)", price: 4 },
            { id: 3, name: "P3", category: "Running",
                description: "P3 (Running)", price: 5 },
            { id: 4, name: "P4", category: "Chess",
                description: "P4 (Chess)", price: 6 },
        ]);
}

```

```

        { id: 5, name: "P5", category: "Chess",
          description: "P6 (Chess)", price: 7 },
      ]);
}

storeOrder(): Promise<number> {
  console.log("Store Order");
  console.log(JSON.stringify(this.order));
  return Promise.resolve(1);
}
}

```

Этот класс использует метод `Promise.resolve` для создания `Promise`, который немедленно выдает ответ и позволяет легко использовать тестовые данные. В главе 17 будет представлен источник данных, который выполняет реальные фоновые операции для запроса данных из веб-сервиса. Чтобы проверить работоспособность основных функций модели данных, замените код в файле `index.ts` кодом из листинга 16.18.

Листинг 16.18. Замена содержимого файла `index.ts` из папки `src`

```

import { LocalDataSource } from "./data/localDataSource";

async function displayData(): Promise<string> {
  let ds = new LocalDataSource();
  let allProducts = await ds.getProducts("name");
  let categories = await ds.getCategories();
  let chessProducts = await ds.getProducts("name", "Chess");

  let result = "";
  allProducts
    .forEach(p => result += `Product: ${p.name}, ${p.category}\n`);
  categories.forEach(c => result += `Category: ${c}\n`);
  chessProducts.forEach(p => ds.order.addProduct(p, 1));
  result += `Order total: $$\{ds.order.total.toFixed(2)}\`;
  return result;
}

displayData().then(res => console.log(res));

```

После сохранения изменений в файле `index.ts` код будет скомпилирован, а цепочка операторов `import` разрешена для включения в пакет webpack всех JavaScript, необходимых приложению. Браузер перезагрузится, и в его JavaScript-консоли отобразится следующий вывод:

```

Product: P1, Watersports
Product: P2, Watersports
Product: P3, Running
Product: P4, Chess
Product: P5, Chess
Category: Watersports
Category: Running
Category: Chess
Order total: $13.00

```

16.6. ОТОБРАЖЕНИЕ HTML-КОНТЕНТА С ИСПОЛЬЗОВАНИЕМ DOM API

Немногие пользователи захотят заглядывать в консоль браузера, чтобы увидеть вывод. Браузеры предоставляют API Domain Object Model (DOM), позволяющий приложениям взаимодействовать с HTML-документом, отображаемым пользователю, динамически генерировать содержимое и реагировать на взаимодействие с пользователем. Чтобы создать класс, который будет создавать HTML-элемент, добавьте в папку `src` файл `domDisplay.ts` и используйте его для определения класса, показанного в листинге 16.19.

Листинг 16.19. Содержимое файла `domDisplay.ts` из папки `src`

```
import { Product, Order } from "./data/entities";

export class DomDisplay {

    props: {
        products: Product[],
        order: Order
    }

    getContent(): HTMLElement {
        let elem = document.createElement("h3");
        elem.innerText = this.getElementText();
        elem.classList.add("bg-primary", "text-center",
        "text-white", "p-2");
        return elem;
    }

    getElementText() {
        return `${this.props.products.length} Products,
            + 'Order total: ${this.props.order.total}`;
    }
}
```

Класс `DomDisplay` определяет метод `getContent`, результатом работы которого является объект `HTMLElement` — тип, применяемый DOM API для представления HTML-элемента. Метод `getContent` создает элемент `H3` и использует шаблонную строку для задания его содержимого. Элемент добавляется в четыре класса, которые должны управлять внешним видом элемента при его отображении. Значения данных, используемых в шаблонной строке, предоставляются через свойство `props`. Это соглашение было заимствовано из фреймворка React.

16.6.1. Добавление поддержки стилей CSS Bootstrap

Три класса, к которым отнесен элемент `h3` в листинге 16.19, соответствуют стилям, определенным в Bootstrap. Bootstrap — это высококачественный CSS-фреймворк с открытым исходным кодом, который упрощает создание единообразного стиля HTML-содержимого.

Конфигурацию webpack можно расширить с помощью загрузчиков для дополнительных типов контента, включенных в файл пакета. Это значит, что инструментарий разработки можно расширить для поддержки таблиц стилей CSS, например для той, которая определяет стили Bootstrap, применяемые к элементу `h3`.

Остановите процесс WDS нажатием `Control+C` и выполните в папке `webapp` команды, показанные в листинге 16.20, чтобы установить загрузчики CSS и пакеты Bootstrap.

ПРИМЕЧАНИЕ

Я использую CSS-фреймворк Bootstrap в большинстве своих проектов, поскольку с ним легко работать и он дает хорошие результаты. Подробную информацию о доступных стилях и дополнительных возможностях JavaScript см. на сайте <https://getbootstrap.com>.

Листинг 16.20. Добавление пакетов в проект

```
npm install bootstrap@5.2.3
npm install --save-dev css-loader@6.7.3
npm install --save-dev style-loader@3.3.2
```

Пакет `bootstrap` содержит стили CSS, необходимые проекту примера. Пакеты `css-loader` и `style-loader` содержат загрузчики, работающие со стилями CSS (оба необходимы для включения CSS в пакет webpack). Внесите изменения, показанные в листинге 16.21, в конфигурацию webpack, чтобы добавить поддержку включения CSS в файл пакета.

Листинг 16.21. Добавление загрузчика в файл webpack.config.js из папки webapp

```
module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: "./src/index.ts",
  output: { filename: "bundle.js" },
  resolve: { extensions: [".ts", ".js", ".css"] },
  module: {
    rules: [
      { test: /\.ts/, use: "ts-loader", exclude: /node_modules/ },
      { test: /\.css$/, use: ["style-loader", "css-loader"] }
    ]
  },
  devServer: {
    static: "./assets",
    port: 4500
  }
};
```

В листинге 16.22 был изменен код в файле `index.ts`, чтобы объявить зависимость от таблицы стилей CSS из пакета Bootstrap и использовать класс `DomHeader` для отображения HTML-содержимого в браузере.

Листинг 16.22. Отображение HTML-содержимого в файле index.ts из папки src

```
import { LocalDataSource } from "./data/localDataSource";
import { DomDisplay } from "./domDisplay";
import "bootstrap/dist/css/bootstrap.css";

let ds = new LocalDataSource();

async function displayData(): Promise<HTMLElement> {
    let display = new DomDisplay();
    display.props = {
        products: await ds.getProducts("name"),
        order: ds.order
    }
    return display.getContent();
}

document.onreadystatechange = () => {
    if (document.readyState === "complete") {
        displayData().then(elem => {
            let rootElement = document.getElementById("app");
            rootElement.innerHTML = "";
            rootElement.appendChild(elem);
        });
    }
};
```

DOM API предоставляет полный набор функций для работы с HTML-документом, отображаемым браузером, но в результате может получиться громоздкий код, который трудно читать, особенно когда отображаемый контент зависит от результатов выполнения фоновых задач, таких как получение данных из веб-сервиса.

Код в листинге 16.22 ожидает выполнения двух задач, прежде чем он сможет отобразить какой-либо контент. Браузер должен завершить обработку HTML-документа, содержащегося в файле `index.html`, до того, как можно будет использовать DOM API для работы с его содержимым. Браузеры обрабатывают элементы HTML в том порядке, в котором они определены в HTML-документе, то есть код JavaScript будет выполнен до того, как браузер обработает элементы в секции `body` документа. Любая попытка изменить документ до его полной обработки рискует привести к противоречивым результатам.

СОВЕТ

В настройках по умолчанию для компилятора TypeScript включены файлы деклараций типов для DOM API, что позволяет использовать возможности браузера типобезопасным образом.

Код в листинге 16.22 также должен ждать, пока источник данных получит свои данные. Класс `LocalDataSource` использует локальные тестовые данные, которые

доступны сразу, но может возникнуть задержка при получении данных из веб-сервиса, который мы реализуем в главе 17.

После выполнения обеих задач элемент-заполнитель в файле `index.html` удаляется и заменяется объектом `HTMLElement`, полученным путем создания объекта `DomDisplay` и вызова его метода `getContent`.

Сохраните изменения в файле `index.ts` и в папке `webapp` выполните команду из листинга 16.23 для запуска сервера разработки Webpack с использованием конфигурации, созданной в листинге 16.21.

Листинг 16.23. Запуск средств разработки

```
npx webpack serve
```

Будет создан новый пакет, включающий стили CSS. С помощью браузера перейдите по адресу `http://localhost:4500`, и на экране появится стилизованное HTML-содержимое, как показано на рис. 16.5.



Рис. 16.5. Генерация элементов HTML

СОВЕТ

Загрузчики, добавленные в проект, работают с CSS путем добавления JavaScript-кода, который выполняется при обработке содержимого файла пакета. Этот код использует API, предоставляемый браузером, для создания стилей CSS. Благодаря такому подходу файл пакета содержит только JavaScript, даже если он доставляет разные типы содержимого на клиентскую сторону.

16.7. ИСПОЛЬЗОВАНИЕ JSX ДЛЯ СОЗДАНИЯ HTML-КОНТЕНТА

Выражать HTML-элементы с помощью операторов JavaScript неудобно, а использование DOM API напрямую приводит к появлению объемного, трудного для понимания и подверженного ошибкам кода, даже с учетом поддержки статических типов, которую обеспечивает TypeScript.

Проблема заключается не в самом DOM API – хотя он не всегда разрабатывался с упором на простоту использования, – а в сложности использования операторов кода для создания декларативного контента, например элементов HTML. Более элегантный подход заключается в применении *JavaScript XML (JSX)*, который позволяет легко смешивать декларативный контент, такой как HTML-элементы,

с операторами кода. JSX наиболее тесно связан с разработкой React, однако компилятор TypeScript позволяет использовать его в любом проекте.

ПРИМЕЧАНИЕ

JSX — не единственный способ упростить работу с элементами HTML, но я использовал его в этой главе, поскольку компилятор TypeScript поддерживает его. Если вам не нравится JSX, воспользуйтесь одним из множества доступных пакетов шаблонов JavaScript (начните с пакета шаблонов mustache).

Лучший способ понять JSX — начать с написания кода JSX. Файлы TypeScript с JSX-содержимым имеют расширение `tsx`, что отражает комбинацию возможностей TypeScript и JSX. Добавьте в папку `src` файл `htmlDisplay.tsx` и поместите в него содержимое, показанное в листинге 16.24.

Листинг 16.24. Содержимое файла `htmlDisplay.tsx` из папки `src`

```
import { Product, Order } from "./data/entities";

export class HtmlDisplay {

    props: {
        products: Product[],
        order: Order
    }

    getContent(): HTMLElement {
        return <h3 className="bg-secondary text-center text-white p-2">
            { this.getElementText() }
        </h3>
    }

    getElementText() {
        return `${this.props.products.length} Products,
            + 'Order total: ${this.props.order.total}'`;
    }
}
```

Этот файл использует JSX для создания того же результата, что и обычный класс TypeScript. Отличие лишь в методе `getContent`, который возвращает HTML-элемент, выраженный непосредственно как элемент, вместо того чтобы использовать DOM API для создания объекта и его настройки через свойства. Возвращаемый элемент `h3` выражается аналогично элементу в HTML-документе, с добавлением фрагментов JavaScript, позволяющих выражениям динамически генерировать контент на основе значений, предоставляемых через свойство `props`.

Данный файл не скомпилируется, поскольку проект еще не настроен на JSX, но вы можете увидеть, как этот формат будет использован для создания более естественного контента. В последующих разделах вы узнаете, как обрабатываются JSX-файлы, и самостоятельно настройте пример проекта на их поддержку.

16.7.1. Как устроен рабочий процесс с JSX

Когда TypeScript JSX-файл компилируется, компилятор обрабатывает содержащиеся в нем HTML-элементы, преобразуя их в операторы JavaScript. Каждый элемент анализируется и разделяется на три части: тег, который определяет тип элемента, атрибуты, применяемые к элементу, и содержимое элемента.

Компилятор заменяет каждый HTML-элемент вызовом функции, известной как *фабричная функция*, отвечающая за создание HTML-контента во время выполнения. Фабричная функция условно называется `createElement`, поскольку именно такое имя используется во фреймворке React. Класс в листинге 16.24 преобразуется в такой код:

```
...
import { Product, Order } from "./data/entities";

export class HtmlDisplay {

    props: {
        products: Product[],
        order: Order
    }

    getContent() {
        return createElement("h3",
            { className: "bg-secondary text-center text-white p-2" },
            this.getElementText());
    }

    getElementText() {
        return `${this.props.products.length} Products,
            + 'Order total: ${this.props.order.total}`;
    }
}
...

```

Компилятор ничего не знает о фабричной функции, кроме ее имени. В результате преобразования HTML-содержимое заменяется операторами кода, которые можно скомпилировать и выполнять с помощью обычной среды выполнения JavaScript (рис. 16.6).

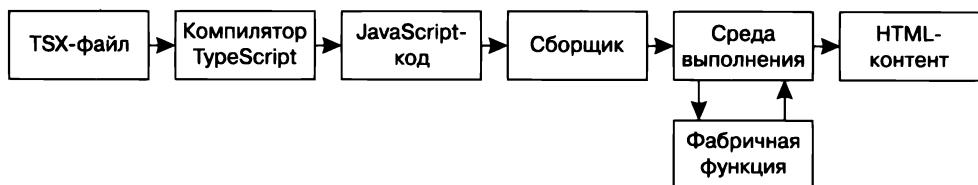


Рис. 16.6. Преобразование JSX

При запуске приложения каждый вызов фабричной функции отвечает за использование проанализированных компилятором имени тега, атрибута и содержимого для создания требуемого приложением HTML-элемента.

ПОНЯТИЕ РЕКВИЗИТОВ И АТРИБУТОВ

Элементы JSX-файла не являются стандартным HTML. Основное отличие в том, что атрибуты элементов используют имена свойств JavaScript, определенных DOM API, вместо соответствующих имен атрибутов из спецификации HTML. Многие свойства и атрибуты обладают одинаковыми именами, но есть и некоторые важные различия. Наибольшую путаницу вызывает атрибут `class`, который используется для назначения элементов одному или нескольким классам, обычно для применения стилей.

DOM API не может использовать `class`, так как это зарезервированное слово в JavaScript, поэтому элементы присваиваются классам с помощью свойства `className`, например, так:

```
...<h3 className="bg-secondary text-center text-white p-2">
...
```

Именно по этой причине JSX-классы TypeScript получают значения своих данных через свойство с именем `props`, поскольку каждый реквизит (`prop`) соответствует свойству, которое должно быть установлено у объекта `HTMLElement`, созданного фабричной функцией. Забыть использовать имена свойств в JSX-файле — распространенная ошибка, и с этого стоит начать проверку, если вы не получаете ожидаемых результатов.

16.7.2. Настройка компилятора и загрузчика

Компилятор TypeScript по умолчанию не обрабатывает файлы TSX и требует установки двух параметров конфигурации, описанных в табл. 16.2. Существуют и другие параметры компилятора для JSX, но для начала работы необходимы именно эти два.

Таблица 16.2. Настройки компилятора для JSX

Название	Описание
<code>jsx</code>	Указывает TypeScript, что файлы с расширением <code>tsx</code> содержат JSX-код. При задании параметра <code>react</code> элементы HTML заменяются вызовами фабричной функции и создается JavaScript-файл. Параметр <code>react-native</code> генерирует JavaScript-файл, в котором HTML-элементы остаются нетронутыми. Настройка <code>preserve</code> формирует JSX-файл, также оставляя HTML-элементы нетронутыми. Настройка <code>react-jsx</code> использует <code>__jsx</code> в качестве имени функции, создающей элементы
<code>jsxFactory</code>	Задает имя фабричной функции, которую компилятор будет использовать, если опция <code>jsx</code> имеет значение <code>react</code>

Для этого проекта мы определим фабричную функцию `createElement` и выберем опцию `react` для параметра `jsx`, чтобы компилятор заменил HTML-содержимое вызовами фабричной функции, как показано в листинге 16.25.

Листинг 16.25. Настройка компилятора в файле tsconfig.json из папки webapp

```
{
  "compilerOptions": {
    "target": "es2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "jsx": "react",
    "jsxFactory": "createElement"
  }
}
```

Конфигурация webpack должна быть обновлена таким образом, чтобы файлы TSX включались в процесс создания пакетов, как показано в листинге 16.26.

Листинг 16.26. Настройка webpack в файле webpack.config.js из папки webapp

```
module.exports = {
  mode: "development",
  devtool: "inline-source-map",
  entry: "./src/index.ts",
  output: { filename: "bundle.js" },
  resolve: { extensions: [".ts", ".tsx", ".js", ".css"] },
  module: {
    rules: [
      { test: /\.tsx?$/, use: "ts-loader", exclude: /node_modules/ },
      { test: /\.css$/, use: ["style-loader", "css-loader"] }
    ]
  },
  devServer: {
    static: "./assets",
    port: 4500
  }
};
```

Изменение параметра `resolve` сообщает webpack, что файлы TSX должны быть включены в пакет, а другое изменение указывает, что файлы TSX будут обрабатываться пакетом `ts-loader`, который использует компилятор TypeScript.

16.7.3. Создание фабричной функции

Код, генерируемый компилятором, заменяет HTML-контент вызовами фабричной функции, что позволяет преобразовать JSX-код в стандартный JavaScript. Реализация фабричной функции зависит от среды, в которой выполняется приложение, так что, например, приложения React будут использовать фабричную функцию, которая генерирует контент, управляемый React. В качестве примера мы создадим фабричную функцию, которая просто применяет DOM API для создания объекта `HTMLElement`. Это далеко не так элегантно и эффективно, как обработка динамического содержимого с помощью React и других фреймворков, но этого достаточно, чтобы использовать JSX в приложении, не погружаясь в детали. Для определения фабричной функции создайте папку `src/tools` и поместите в нее файл с именем `jsxFactory.ts` с кодом из листинга 16.27.

Листинг 16.27. Содержимое файла `jsxFactory.ts` из папки `src/tools`

```

export function createElement(tag: any, props: Object,
    ...children : Object[] ) : HTMLElement {

    function addChild(elem: HTMLElement, child: any) {
        elem.appendChild(child instanceof Node ? child
            : document.createTextNode(child.toString()));
    }

    if (typeof tag === "function") {
        return Object.assign(new tag(), { props: props || {} }).getContent();
    }

    const elem = Object.assign(document.createElement(tag), props || {});
    children.forEach(child => Array.isArray(child)
        ? child.forEach(c => addChild(elem, c)) : addChild(elem, child));
    return elem;
}

declare global {
    namespace JSX {
        interface ElementAttributesProperty { props; }
    }
}

```

Функция `createElement` из листинга 16.27 выполняет необходимый минимум действий по созданию HTML-элементов с применением DOM API без каких-либо сложных возможностей, предоставляемых фреймворками. Параметр `tag` может быть функцией, в этом случае в качестве типа элемента указан другой класс, использующий JSX.

СОВЕТ

Последний участок кода в листинге 16.27 представляет собой своеобразное «заклинание», указывающее компилятору TypeScript, что ему следует использовать свойство `props` для проверки типов значений, присвоенных атрибутам элементов JSX в файлах TSX. Это опирается на функциональность пространства имен TypeScript, которую я не описывал в этой главе, поскольку она была заменена стандартными модулями JavaScript и более не рекомендуется к использованию.

16.7.4. Использование класса JSX

Классы JSX преобразуются в стандартный код JavaScript, что позволяет использовать их так же, как и любой класс TypeScript. В листинге 16.28 убрана зависимость от класса DOM API, который заменен классом JSX.

Листинг 16.28. Использование JSX-класса в файле `index.ts` из папки `src`

```

import { LocalDataSource } from "./data/localDataSource";
import { HtmlDisplay } from "./htmlDisplay";
import "bootstrap/dist/css/bootstrap.css";

let ds = new LocalDataSource();

```

```

async function displayData(): Promise<HTMLElement> {
    let display = new HtmlDisplay();
    display.props = {
        products: await ds.getProducts("name"),
        order: ds.order
    }
    return display.getContent();
}

document.onreadystatechange = () => {
    if (document.readyState === "complete") {
        displayData().then(elem => {
            let rootElement = document.getElementById("app");
            rootElement.innerHTML = "";
            rootElement.appendChild(elem);
        });
    }
};

```

Класс JSX является полноценной заменой класса, который напрямую использует DOM API. В последующих разделах будет показано, как классы с JSX можно комбинировать с помощью только элементов. Однако необходимо помнить, что всегда существует граница между обычным классом и классом, содержащим HTML-элементы. В нашем примере эта граница проходит между файлом `index` и классом `HtmlDisplay`.

16.7.5. Импорт фабричной функции в класс JSX

Последнее изменение для завершения настройки JSX заключается в добавлении оператора `import` для фабричной функции в класс JSX, как показано в листинге 16.29. Компилятор TypeScript преобразует HTML-элементы в вызовы фабричной функции, но для компиляции преобразованного кода нужен оператор `import`.

Листинг 16.29. Добавление оператора импорта в файл `htmlDisplay.tsx` из папки `src`

```

import { createElement } from "./tools/jsxFactory";
import { Product, Order } from "./data/entities";

export class HtmlDisplay {

    props: {
        products: Product[],
        order: Order
    }

    getContent(): HTMLElement {
        return <h3 className="bg-secondary text-center text-white p-2">
            { this.getElementText() }
        </h3>
    }

    getElementText() {
        return `${this.props.products.length} Products,
            + Order total: ${this.props.order.total}`;
    }
}

```

В каждом файле TSX требуется оператор `import` для фабричной функции. С помощью **Control+C** остановите работу средств разработки webpack и, используя командную строку, выполните команду из листинга 16.30 в папке `webapp`, чтобы запустить их снова с новой конфигурацией.

Листинг 16.30. Запуск средств разработки

```
npm webpack serve
```

После повторного создания пакета перейдите с помощью браузера по адресу `http://localhost:4500`, и вы увидите содержимое (рис. 16.7), оформленное другим цветом по сравнению с предыдущим примером.



Рис. 16.7. Рендеринг содержимого с использованием JSX

16.8. ДОБАВЛЕНИЕ ФУНКЦИЙ В ПРИЛОЖЕНИЕ

Теперь, когда базовая структура приложения создана, мы можем добавить функции, начиная с отображения товаров с возможностью фильтрации по категориям.

16.8.1. Отображение отфильтрованного списка товаров

Добавьте файл `productItem.tsx` в папку `src` и поместите в него код из листинга 16.31 для создания класса, в котором будут отображаться сведения об одном товаре.

Листинг 16.31. Содержимое файла `productItem.tsx` из папки `src`

```
import { createElement } from "./tools/jsxFactory";
import { Product } from "./data/entities";

export class ProductItem {
    private quantity: number = 1;

    props: {
        product: Product,
        callback: (product: Product, quantity: number) => void
    }

    getContent(): HTMLElement {
        return <div className="card card-outline-primary m-1 p-1 bg-light">
            <h4>
                { this.props.product.name }
            <span className="badge rounded-pill bg-primary text-white" style="float:right">
                <small>${ this.props.product.price.toFixed(2) }</small>
            </span>
        </div>
    }
}
```

```

        </h4>
        <div className="card-text bg-white p-1">
            { this.props.product.description }
            <button className="btn btn-success btn-sm float-end"
                onClick={ this.handleAddToCart } >
                Add To Cart
            </button>
            <select className="form-control-inline float-end m-1"
                onChange={ this.handleQuantityChange }>
                <option>1</option>
                <option>2</option>
                <option>3</option>
            </select>
        </div>
    </div>
}

handleQuantityChange = (ev: Event): void => {
    this.quantity = Number((ev.target as HTMLSelectElement).value);
}

handleAddToCart = (): void => {
    this.props.callback(this.props.product, this.quantity);
}
}

```

Класс `ProductItem` принимает объект `Product` и функцию обратного вызова через реквизиты. Метод `getContent` выводит HTML-элементы, отображающие информацию об объекте `Product`, а также элемент `select`, позволяющий выбрать количество товара, и кнопку `button`, нажав на которую пользователь добавит товар в заказ.

Элементы `select` и `button` настраиваются с помощью функции обработки событий, используя реквизиты `onChange` и `onClick`. Методы, обрабатывающие события, определяются синтаксисом стрелочной функции, например, так:

```

...
handleQuantityChange = (ev: Event): void => {
    this.quantity = Number((ev.target as HTMLSelectElement).value);
}
...

```

Синтаксис стрелочной функции гарантирует, что ключевое слово `this` ссылается на объект `ProductItem`, что позволяет использовать свойства `props` и `quantity`. Если для обработки события используется обычный метод, то `this` ссылается на объект, описывающий это событие.

Декларации типов в TypeScript для обработки событий DOM API неудобны и требуют утверждения типа для цели события перед получением доступа к его функциям.

```

...
handleQuantityChange = (ev: Event): void => {
    this.quantity = Number((ev.target as HTMLSelectElement).value);
}
...

```

Чтобы прочитать свойство `value` из элемента `select`, необходимо применить утверждение, сообщающее компилятору TypeScript, что свойство `event.target` вернет объект `HTMLSelectElement`.

СОВЕТ

Тип `HTMLSelectElement` является одним из стандартных типов DOM API, которые подробно описаны на сайте <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>.

Для отображения списка кнопок категорий, позволяющих пользователю фильтровать контент, добавьте в папку `src` файл `categoryList.tsx` с содержимым, показанным в листинге 16.32.

Листинг 16.32. Содержимое файла `categoryList.tsx` из папки `src`

```
import { createElement } from "./tools/jsxFactory";

export class CategoryList {

    props: {
        categories: string[];
        selectedCategory: string,
        callback: (selected: string) => void
    }

    getContent(): HTMLElement {
        return <div className="d-grid gap-2">
            { ["All", ...this.props.categories]
                .map(c => this.getCategoryButton(c))}
        </div>
    }

    getCategoryButton(cat?: string): HTMLElement {
        let selected = this.props.selectedCategory === undefined
            ? "All": this.props.selectedCategory;
        let btnClass = selected === cat ? "btn-primary": "btn-secondary";
        return <button className={ 'btn btn-block ${btnClass}' }
            onClick={ () => this.props.callback(cat)}>
            { cat }
        </button>
    }
}
```

Этот класс отображает список элементов `button`, стилизованных с использованием классов Bootstrap. Реквизиты этого класса содержат список категорий, для которых следует создать кнопки, текущую выбранную категорию и функцию обратного вызова, которая вызывается при нажатии пользователем на кнопку:

```
...
return <button className={ 'btn btn-block ${btnClass}' }
    onClick={ () => this.props.callback(cat) }>
...

```

Данный шаблон характерен при использовании JSX, поэтому классы отображают HTML-элементы, используя данные, полученные через реквизиты.

Эти реквизиты также включают функции обратного вызова, которые реагируют в ответ на события. В нашем случае атрибут `onclick` необходим для вызова функции, полученной через реквизит `callback`.

Для отображения списка товаров и кнопок категорий добавьте в папку `src` файл с именем `productList.tsx` с содержимым, показанным в листинге 16.33.

Листинг 16.33. Содержимое файла `productList.tsx` из папки `src`

```
import { createElement } from "./tools/jsxFactory";
import { Product } from "./data/entities";
import { ProductItem } from "./productItem";
import { CategoryList } from "./categoryList";

export class ProductList {
    props: {
        products: Product[],
        categories: string[],
        selectedCategory: string,
        addToOrderCallback?: (product: Product, quantity: number) => void,
        filterCallback?: (category: string) => void;
    }

    getContent(): HTMLElement {
        return <div className="container-fluid">
            <div className="row">
                <div className="col-3 p-2">
                    <CategoryList categories={ this.props.categories }
                        selectedCategory={ this.props.selectedCategory }
                        callback={ this.props.filterCallback } />
                </div>
                <div className="col-9 p-2">
                    {
                        this.props.products.map(p =>
                            <ProductItem product={ p }
                                callback={ this.props.addToOrderCallback } />
                        )
                    }
                </div>
            </div>
        </div>
    }
}
```

Метод `getContent` в этом классе опирается на одну из самых полезных функций JSX — возможность применять другие JSX-классы в качестве HTML-элементов, как это сделано здесь:

```
...
<div className="col-3 p-2">
    <CategoryList categories={ this.props.categories }
        selectedCategory={ this.props.selectedCategory }
        callback={ this.props.filterCallback } />
</div>
...
```

При анализе TSX-файла компилятор TypeScript обнаруживает, что пользовательский тег создает оператор,зывающий фабричную функцию с соответствующим

классом. Во время выполнения создается новый экземпляр класса, атрибуты элемента присваиваются свойству `props` и вызывается метод `getContent`, чтобы получить содержимое для включения в представляемый пользователю HTML.

16.8.2. Отображение содержимого и обработка обновлений

Нам необходимо создать мост между функциональностью хранилища данных и классами JSX, которые отображают контент пользователю, обеспечивая обновление контента с учетом изменений в состоянии приложения. Фреймворки, демонстрируемые в последующих главах, обеспечат эффективную обработку обновлений и минимизируют объем работы, выполняемой браузером по отображению изменений.

Для данного примера воспользуемся самым простым подходом, который заключается в том, что изменения происходят путем уничтожения и повторного создания HTML-элементов, отображаемых браузером, как показано в листинге 16.34, где класс `HtmlDisplay` переделан таким образом, что он получает источник данных и управляет данными состояния, необходимыми для отображения списка товаров, отфильтрованных по категориям.

Листинг 16.34. Отображение содержимого в файле `htmlDisplay.tsx` из папки `src`

```
import { createElement } from "./tools/jsxFactory";
import { Product, Order } from "./data/entities";
import { AbstractDataSource } from "./data/abstractDataSource";
import { ProductList } from "./productList";

export class HtmlDisplay {
    private containerElem: HTMLElement;
    private selectedCategory: string;

    constructor() {
        this.containerElem = document.createElement("div");
    }

    props: {
        dataSource: AbstractDataSource;
    }

    async getContent(): Promise<HTMLElement> {
        await this.updateContent();
        return this.containerElem;
    }

    async updateContent() {
        let products = await this.props.dataSource.getProducts("id",
            this.selectedCategory);
        let categories = await this.props.dataSource.getCategories();
        this.containerElem.innerHTML = "";
        let content = <div>
            <ProductList products={ products } categories={ categories }>
                <selectedCategory={ this.selectedCategory }>
                    <addToOrderCallback={ this.addToOrder }>
            </ProductList>
        </div>
    }
}
```

```

        filterCallback={ this.selectCategory } />
    </div>
    this.containerElem.appendChild(content);
}

addToOrder = (product: Product, quantity: number) => {
    this.props.dataSource.order.addProduct(product, quantity);
    this.updateContent();
}

selectCategory = (selected: string) => {
    this.selectedCategory = selected === "All" ? undefined : selected;
    this.updateContent();
}
}

```

Методы, определенные классом `HtmlDisplay`, используются в качестве функций обратного вызова для класса `ProductList`, который передает их классам `ProductItem` и `CategoryList`. При вызове этих методов происходит обновление свойств, отслеживающих состояние приложения, а затем вызывается метод `updateContent`, который заменяет HTML, отображаемый классом.

Чтобы предоставить классу `HtmlDisplay` необходимые реквизиты, обновите файл `index.ts`, как показано в листинге 16.35.

Листинг 16.35. Изменение реквизитов в файле index.ts из папки src

```

import { LocalDataSource } from "./data/localDataSource";
import { HtmlDisplay } from "./htmlDisplay";
import "bootstrap/dist/css/bootstrap.css";

let ds = new LocalDataSource();

async function displayData(): Promise<HTMLElement> {
    let display = new HtmlDisplay();
    display.props = {
        // products: await ds.getProducts("name"),
        // order: ds.order
        dataSource: ds
    }
    return display.getContent();
}

document.onreadystatechange = () => {
    if (document.readyState === "complete") {
        displayData().then(elem => {
            let rootElement = document.getElementById("app");
            rootElement.innerHTML = "";
            rootElement.appendChild(elem);
        });
    }
};

```

При сохранении изменений будет создан новый пакет, который приведет к перезагрузке браузера, и на экране отобразится содержимое, как на рис. 16.8. Как видно из рисунка, при нажатии на кнопку категории происходит фильтрация товаров, отображаемых пользователю.

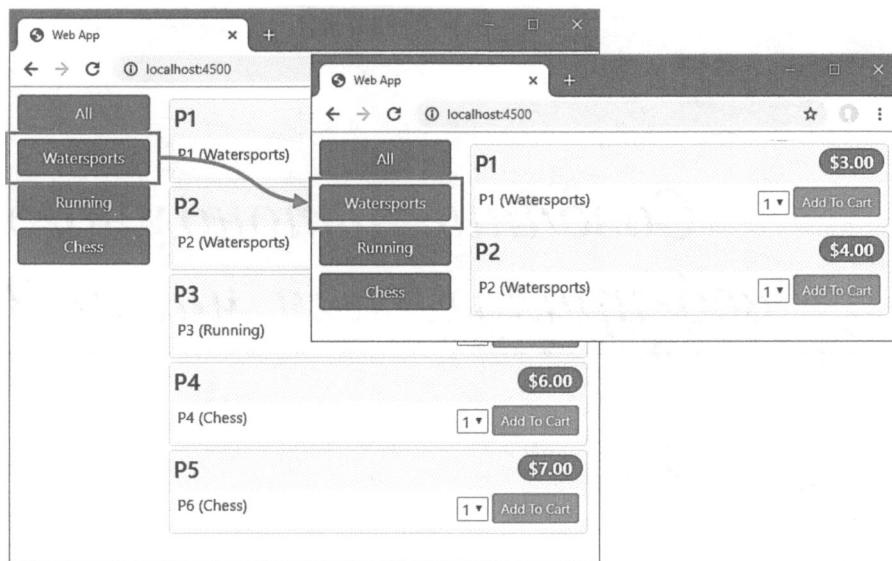


Рис. 16.8. Отображение товаров

РЕЗЮМЕ

В этой главе вы научились создавать простую, но эффективную цепочку инструментов для разработки веб-приложений, используя компилятор TypeScript и webpack. Вы узнали, как вывод компилятора TypeScript можно включить в пакет webpack, а также как использовать поддержку JSX для упрощения работы с HTML-элементами.

- Сборщики, или бандлеры, – это инструменты, объединяющие ресурсы проекта в файлы, которые можно эффективно доставить в браузер.
- JSX – это формат файлов, объединяющий код и разметку, что облегчает создание контента для веб-приложения.
- Элементы JSX не являются стандартными элементами HTML, и для того, чтобы избежать использования зарезервированных ключевых слов JavaScript, таких как `class`, были внесены соответствующие изменения.
- Фабричная функция используется для преобразования содержимого JSX в JavaScript-код. Обычно фабричные функции предоставляются фреймворком веб-приложения, но можно использовать и собственные.
- Содержимое, генерируемое из JSX-файлов, может включать стили CSS, при этом таблицы стилей включаются в пакеты, доставляемые в браузер.

В следующей главе мы создадим финальную версию автономного веб-приложения и подготовим его к развертыванию.

Создание автономного веб-приложения, часть 2

В этой главе

- ✓ Создание и потребление веб-сервиса.
- ✓ Завершение работы с основными функциями приложения.
- ✓ Создание сервера развертывания и постоянного хранилища данных.
- ✓ Разворачивание приложения в контейнере.

В этой главе я завершаю создание автономного веб-приложения и готовлю его к развертыванию, демонстрируя, как проект на TypeScript сочетается со стандартными процессами разработки для развертывания. В качестве краткой справки в табл. 17.1 перечислены параметры компилятора TypeScript, используемые в этой главе.

Таблица 17.1. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
jsx	Определяет, как обрабатываются HTML-элементы в файлах JSX/TSX
jsxFactory	Задает имя фабричной функции для замены HTML-элементов в файлах JSX/TSX
moduleResolution	Задает стиль разрешения модуля, который следует использовать для разрешения зависимостей

Название	Описание
outDir	Задает каталог, в который будут помещены файлы JavaScript
rootDir	Задает корневой каталог, который компилятор будет использовать для поиска файлов TypeScript
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

17.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В данной главе мы продолжим использовать проект, созданный в главе 16. Прежде чем приступить к работе с материалом из этой главы, откройте новое окно командной строки, перейдите в папку `webapp` и выполните команды, показанные в листинге 17.1, для добавления новых пакетов в проект.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 17.1. Добавление пакетов в проект

```
npm install --save-dev json-server@0.17.3
npm install --save-dev npm-run-all@4.1.5
```

Пакет `json-server` представляет собой REST-ориентированный веб-сервис, который будет предоставлять данные для приложения, заменяя локальные тестовые данные, использовавшиеся в главе 16. Пакет `npm-run-all` — полезный инструмент для запуска нескольких пакетов NPM с помощью одной команды.

Чтобы предоставить веб-сервису его данные, создайте в папке `webapp` файл с именем `data.js` и поместите в него код, показанный в листинге 17.2.

Листинг 17.2. Содержимое файла `data.js` из папки `webapp`

```
module.exports = function () {
  return {
    products: [
      { id: 1, name: "Kayak", category: "Watersports",
        description: "A boat for one person", price: 275 },
      { id: 2, name: "Lifejacket", category: "Watersports",
        description: "Protective and fashionable", price: 48.95 },
      { id: 3, name: "Soccer Ball", category: "Soccer",
        description: "FIFA-approved size and weight",
        price: 19.50 },
      { id: 4, name: "Corner Flags", category: "Soccer", description:
        "Give your playing field a professional touch",
        price: 34.95 },
    ]
  }
}
```

```

        { id: 5, name: "Stadium", category: "Soccer",
          description: "Flat-packed 35,000-seat stadium",
          price: 79500 },
        { id: 6, name: "Thinking Cap", category: "Chess",
          description: "Improve brain efficiency by 75%", price: 16 },
        { id: 7, name: "Unsteady Chair", category: "Chess",
          description: "Secretly give your opponent a disadvantage",
          price: 29.95 },
        { id: 8, name: "Human Chess Board", category: "Chess",
          description: "A fun game for the family", price: 75 },
        { id: 9, name: "Bling Bling King", category: "Chess",
          description: "Gold-plated, diamond-studded King",
          price: 1200 }
      ],
      orders: []
    }
  }
}

```

Пакет `json-server` будет настроен на использование данных из листинга 17.2, что приведет к их сбросу при каждом перезапуске. (Пакет также может хранить данные постоянно, но это не так удобно для наших примеров проектов, где более полезна базовая конфигурация.)

Чтобы настроить средства разработки, обновите раздел `scripts` файла `package.json`, как показано в листинге 17.3.

Листинг 17.3. Настройка средств разработки в файле package.json из папки webapp

```

...
  "scripts": {
    "json": "json-server data.js -p 4600",
    "wds": "webpack serve",
    "start": "npm-run-all -p json wds"
  },
...

```

Эти записи позволяют одной командой запустить и веб-сервис, предоставляющий данные, и HTTP-сервер webpack. С помощью командной строки выполните в папке `webapp` команду из листинга 17.4.

Листинг 17.4. Запуск средств разработки

```
npm start
```

Веб-сервис запустится, хотя данные еще не интегрированы в приложение. Чтобы протестировать работу веб-сервиса, перейдите в браузере по адресу `http://localhost:4600/products`. Вы получите ответ, показанный на рис. 17.1.

Файлы TypeScript скомпилируются, будет создан пакет, а HTTP-сервер разработки начнет прослушивать HTTP-запросы. Откройте новое окно браузера и перейдите по адресу `http://localhost:4500`, чтобы увидеть содержимое, как показано на рис. 17.2.

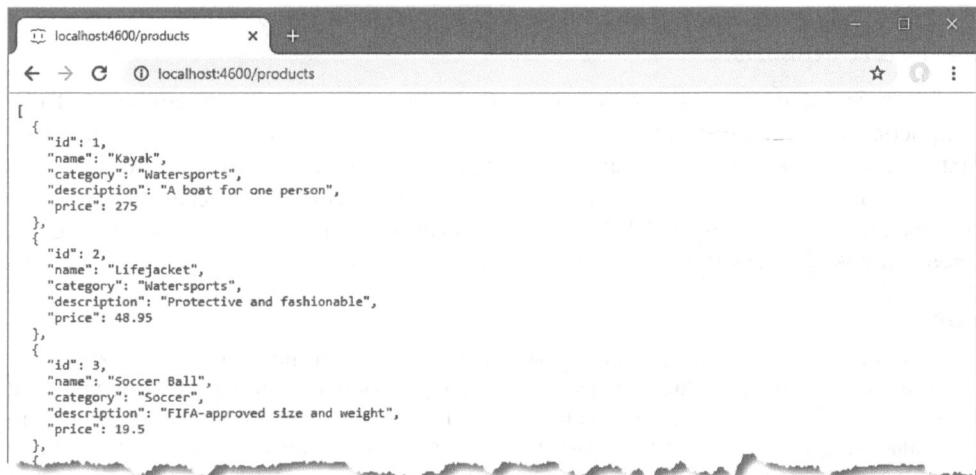


Рис. 17.1. Получение данных из веб-сервиса

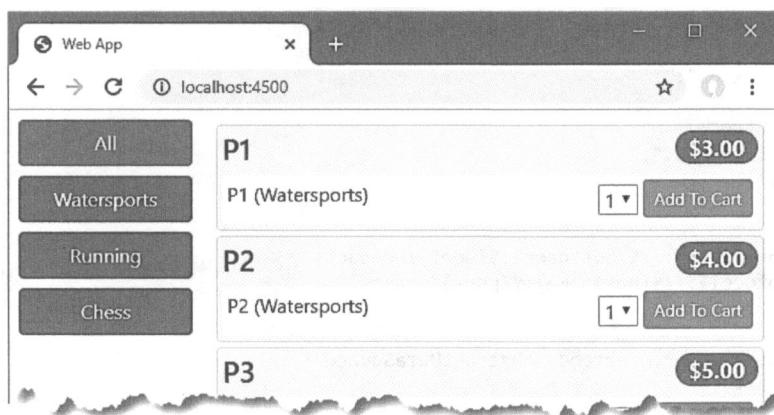


Рис. 17.2. Запуск примера приложения

17.2. ДОБАВЛЕНИЕ ВЕБ-СЕРВИСА

В главе 16 мы использовали локальные тестовые данные для начала работы. Я считаю этот подход полезным для создания фундамента проекта, не увязая в детали получения данных с сервера. Но теперь, когда приложение обретает форму, пришло время добавить веб-сервис и начать работать с удаленными данными. Откройте новое окно командной строки, перейдите в папку `webapp` и выполните команду, показанную в листинге 17.5, чтобы добавить в проект новый пакет.

Листинг 17.5. Добавление пакета в проект

```
npm install axios@1.3.4
```

В JavaScript-приложениях доступно множество пакетов для выполнения HTTP-запросов, все они используют API, предоставляемые браузером. В текущей главе мы воспользуемся популярным пакетом Axios, поскольку с ним легко работать и он поставляется в комплекте с декларациями TypeScript. Для создания источника данных, использующего HTTP-запросы, добавьте в папку `src/data` файл с именем `remoteDataSource.ts` и поместите в него код из листинга 17.6.

СОВЕТ

Существует два API, предоставляемых браузерами, для выполнения HTTP-запросов. Традиционный API — `XMLHttpRequest` — поддерживается всеми браузерами, но с ним сложно работать. Другой API под названием Fetch проще в освоении, но он не поддерживается старыми браузерами. Вы можете использовать любой из них напрямую, но такие пакеты, как Axios, предоставляют API, с которым удобно работать, сохраняя при этом поддержку старых браузеров.

Листинг 17.6. Содержимое файла `remotedatasource.ts` из папки `src/data`

```
import { AbstractDataSource } from "./abstractDataSource";
import { Product, Order } from "./entities";
import Axios from "axios";

const protocol = "http";
const hostname = "localhost";
const port = 4600;

const urls = {
    products: `${protocol}://${hostname}:${port}/products`,
    orders: `${protocol}://${hostname}:${port}/orders`
};

export class RemoteDataSource extends AbstractDataSource {

    loadProducts(): Promise<Product[]> {
        return Axios.get(urls.products).then(response => response.data);
    }

    storeOrder(): Promise<number> {
        let orderData = {
            lines: [...this.order.orderLines.values()].map(ol => ({
                productId: ol.product.id,
                productName: ol.product.name,
                quantity: ol.quantity
            }))
        }
        return Axios.post(urls.orders, orderData)
            .then(response => response.data.id);
    }
}
```

Пакет Axios предоставляет методы `get` и `post`, которые отправляют HTTP-запросы с соответствующими методами. Реализация метода `loadProducts` посылает

GET-запрос к веб-сервису для получения данных о товаре. Метод `storeOrder` преобразует данные о заказе в форму, которую можно легко сохранить, и отправляет их веб-сервису в виде POST-запроса. В ответ на запрос веб-служба вернет сохраненный объект, включающий значение `id`, которое уникально для каждого сохраненного объекта.

17.2.1. Включение источника данных в приложение

Чтобы компилятор TypeScript смог разрешить зависимость от пакета Axios, необходимо изменить конфигурацию, как показано в листинге 17.7.

Листинг 17.7. Настройка компилятора TypeScript в файле `tsconfig.json` из папки `webapp`

```
{  
  "compilerOptions": {  
    "target": "ES2022",  
    "outDir": "./dist",  
    "rootDir": "./src",  
    "jsx": "react",  
    "jsxFactory": "createElement",  
    "moduleResolution": "bundler"  
  }  
}
```

Это изменение сообщает компилятору, что он может разрешать зависимости, просматривая папку `node_modules`, с помощью настройки `bundler`, которая предназначена для работы с такими упаковщиками, как `webpack`. В листинге 17.8 обновлен файл `index.ts` для использования нового источника данных.

Листинг 17.8. Изменение источника данных в файле `index.ts` из папки `src`

```
//import { LocalDataSource } from "./data/localDataSource";  
import { RemoteDataSource } from "./data/remoteDataSource";  
import { HtmlDisplay } from "./htmlDisplay";  
import "bootstrap/dist/css/bootstrap.css";  
  
let ds = new RemoteDataSource();  
  
function displayData(): Promise<HTMLElement> {  
  let display = new HtmlDisplay();  
  display.props = {  
    dataSource: ds  
  }  
  return display.getContent();  
}  
  
document.onreadystatechange = () => {  
  if (document.readyState === "complete") {  
    displayData().then(elem => {  
      let rootElement = document.getElementById("app");  
      rootElement.innerHTML = "";  
      rootElement.appendChild(elem);  
    });  
  }  
};
```

Для применения изменения конфигурации в листинге 17.7 необходимо перезапустить средства разработки. Нажатием Control+C остановите работу веб-сервиса и webpack, а затем в папке `webapp` выполните команду из листинга 17.9, чтобы запустить их снова.

Листинг 17.9. Запуск средств разработки

```
npm start
```

Перейдя по адресу `http://localhost:4500`, вы увидите данные, полученные из веб-сервиса (рис. 17.3).

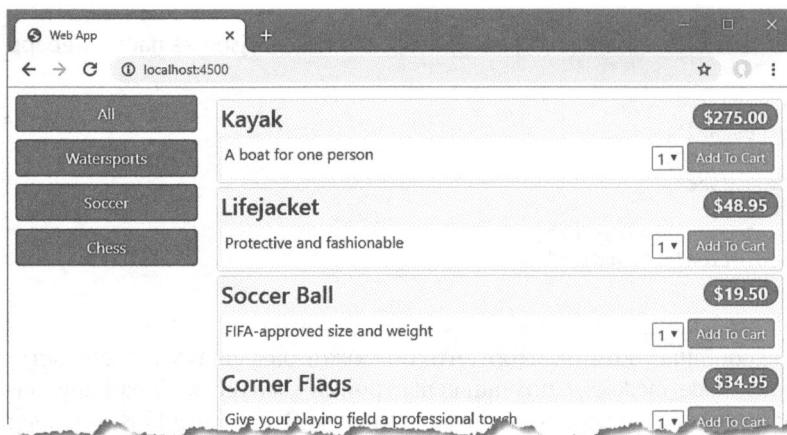


Рис. 17.3. Использование удаленных данных

17.3. ЗАВЕРШЕНИЕ РАБОТЫ НАД ПРИЛОЖЕНИЕМ

Значительная часть главы 16 была посвящена настройке средств разработки и конфигурированию проекта для работы с JSX, который облегчает работу с HTML-содержимым в файлах кода. Теперь, когда базовая структура создана, добавить в приложение новые функции не составит труда. Этот раздел полностью уделен завершению работы над приложением.

17.3.1. Добавление класса заголовка

Чтобы вывести на экран заголовок, содержащий краткую информацию о выбранных пользователем параметрах, добавьте в папку `src` файл `header.tsx` с содержимым, показанным в листинге 17.10.

Листинг 17.10. Содержимое файла `header.tsx` из папки `src`

```
import { createElement } from "./tools/jsxFactory";
import { Order } from "./data/entities";

export class Header {
```

```

props: {
    order: Order,
    submitCallback: () => void
}

getContent(): HTMLElement {
    let count = this.props.order.productCount;
    return <div className="p-1 bg-secondary text-white text-end">
        { count == 0 ? "(No Selection)" :
            `${ count } product(s), $` +
            `${ this.props.order.total.toFixed(2) }` }
        <button className="btn btn-sm btn-primary m-1"
            onClick={ this.props.submitCallback }>
            Submit Order
        </button>
    </div>
}
}

```

Этот класс принимает объект `Order` и функцию обратного вызова через свои реквизиты. На экран выводится простая сводка о заказе `Order`, а также кнопка, при нажатии на которую вызывается функция обратного вызова.

17.3.2. Добавление класса деталей заказа

Для отображения деталей ордера добавьте в папку `src` файл с именем `orderDetails.tsx` и поместите в него код из листинга 17.11.

Листинг 17.11. Содержимое файла `orderDetails.tsx` из папки `src`

```

import { createElement } from "./tools/jsxFactory";
import { Product, Order } from "./data/entities";

export class OrderDetails {

    props: {
        order: Order
        cancelCallback: () => void,
        submitCallback: () => void
    }

    getContent(): HTMLElement {
        return <div>
            <h3 className="text-center bg-primary text-white p-2">
                Order Summary
            </h3>
            <div className="p-3">
                <table className="table table-sm table-striped">
                    <thead>
                        <tr>
                            <th>Quantity</th><th>Product</th>
                            <th className="text-right">Price</th>
                            <th className="text-right">Subtotal</th>
                        </tr>
                    </thead>

```

```

        <tbody>
            { this.props.order.orderLines.map(line =>
                <tr>
                    <td>{ line.quantity }</td>
                    <td>{ line.product.name }</td>
                    <td className="text-right">
                        ${ line.product.price.toFixed(2) }
                    </td>
                    <td className="text-right">
                        ${ line.total.toFixed(2) }
                    </td>
                </tr>
            )}
        </tbody>
        <tfoot>
            <tr>
                <th className="text-right" colSpan="3">
                    Total:
                </th>
                <th className="text-right">
                    ${ this.props.order.total.toFixed(2) }
                </th>
            </tr>
        </tfoot>
    </table>
</div>
<div className="text-center">
    <button className="btn btn-secondary m-1"
        onClick={ this.props.cancelCallback }>
        Back
    </button>
    <button className="btn btn-primary m-1"
        onClick={ this.props.submitCallback }>
        Submit Order
    </button>
</div>
</div>
}
}

```

Класс `OrderDetails` отображает таблицу с деталями заказа, а также кнопки для возврата к списку товаров или отправки заказа.

17.3.3. Добавление класса подтверждения

Чтобы вывести сообщение о том, что заказ был отправлен, добавьте в папку `src` файл с именем `summary.tsx` и поместите в него код из листинга 17.12.

Листинг 17.12. Содержимое файла `summary.tsx` из папки `src`

```

import { createElement } from "./tools/jsxFactory";

export class Summary {

    props: {
        orderId: number,
        callback: () => void
    }
}

```

```
getContent(): HTMLElement {
    return <div className="m-2 text-center">
        <h2>Thanks!</h2>
        <p>Thanks for placing your order.</p>
        <p>Your order is #{ this.props.orderId }</p>
        <p>We'll ship your goods as soon as possible.</p>
        <button className="btn btn-primary"
            onClick={ this.props.callback }>
            OK
        </button>
    </div>
}
```

Этот класс отображает простое сообщение, содержащее уникальный идентификатор, присвоенный веб-сервисом, а также кнопку, которая при нажатии вызывает функцию обратного вызова, полученного в качестве реквизита.

17.3.4. Финальный этап

Последний шаг – добавить код, который объединит классы, созданные в предыдущих разделах, предоставит им необходимые данные и функции обратного вызова через их реквизиты и отобразит сгенерированный ими HTML-контент, как показано в листинге 17.13.

Листинг 17.13. Финальный код приложения в файле htmlDisplay.tsx из папки src

```
import { createElement } from "./tools/jsxFactory";
import { Product, Order } from "./data/entities";
import { AbstractDataSource } from "./data/abstractDataSource";
import { ProductList } from "./productList";
import { Header } from "./header";
import { OrderDetails } from "./orderDetails";
import { Summary } from "./summary";

enum DisplayMode {
    List, Details, Complete
}

export class HtmlDisplay {
    private containerElem: HTMLElement;
    private selectedCategory: string;
    private mode: DisplayMode = DisplayMode.List;
    private orderId: number;

    constructor() {
        this.containerElem = document.createElement("div");
    }

    props: {
        dataSource: AbstractDataSource;
    }

    async getContent(): Promise<HTMLElement> {
        await this.updateContent();
        return this.containerElem;
    }
}
```

```

async updateContent() {
    let products = await this.props.dataSource
        .getProducts("id", this.selectedCategory);
    let categories = await this.props.dataSource.getCategories();
    this.containerElem.innerHTML = "";
    let contentElem: HTMLElement;
    switch (this.mode) {
        case DisplayMode.List:
            contentElem = this.getListContent(products, categories);
            break;
        case DisplayMode.Details:
            contentElem = <OrderDetails
                order={ this.props.dataSource.order }
                cancelCallback={ this.showList }
                submitCallback={ this.submitOrder } />
            break;
        case DisplayMode.Complete:
            contentElem = <Summary orderId={ this.orderId } 
                callback= { this.showList } />
            break;
    }
    this.containerElem.appendChild(contentElem);
}

getListContent(products: Product[], categories: string[])
    : HTMLElement {
    return <div>
        <Header order={ this.props.dataSource.order }
            submitCallback={ this.showDetails } />
        <ProductList products={ products } categories={ categories }
            selectedCategory={ this.selectedCategory }
            addToOrderCallback={ this.addToOrder }
            filterCallback={ this.selectCategory} />
    </div>
}

addToOrder = (product: Product, quantity: number) => {
    this.props.dataSource.order.addProduct(product, quantity);
    this.updateContent();
}

selectCategory = (selected: string) => {
    this.selectedCategory = selected === "All" ? undefined : selected;
    this.updateContent();
}

showDetails = () => {
    this.mode = DisplayMode.Details;
    this.updateContent();
}

showList = () => {
    this.mode = DisplayMode.List;
    this.updateContent();
}

```

```

submitOrder = () => {
  this.props.dataSource.storeOrder().then(id => {
    this.orderId = id;
    this.props.dataSource.order = new Order();
    this.mode = DisplayMode.Complete;
    this.updateContent();
  });
}
}

```

Дополнения к классу `HtmlDisplay` используются для определения того, какие JSX-классы применяются для отображения контента пользователю. Ключевым является свойство `mode`, которое использует значения перечисления `DisplayMode` для выбора содержимого, в сочетании с методами `showDetails`, `showList` и `submitOrder`, которые изменяют значение `mode` и обновляют отображение.

Часто в веб-приложении существует единственный класс, где сосредотачивается вся сложность, даже в таком простом приложении, как это. Использование одного из фреймворков, описанных в последующих главах, может помочь. Однако он просто выражается по-другому, чаще всего в виде сложного набора соответствий между URL-адресами, поддерживаемыми приложением, и классами контента, к которым они относятся.

После сохранения всех изменений и загрузки нового пакета браузером вы сможете выбрать товар, просмотреть его и отправить на сервер, как показано на рис. 17.4.

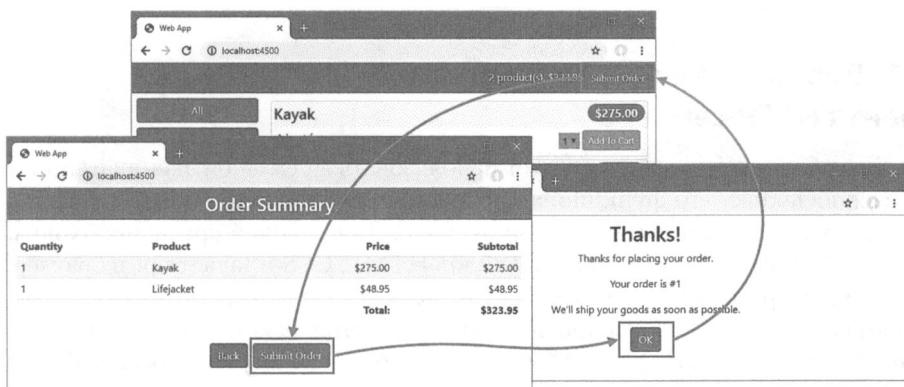
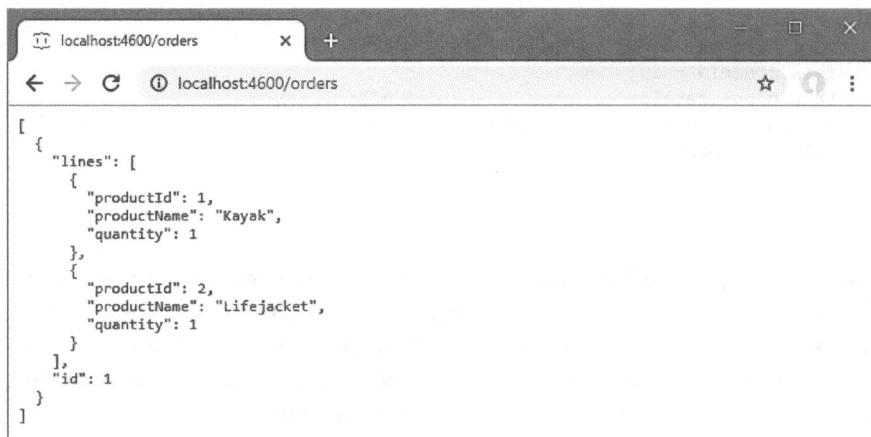


Рис. 17.4. Использование примера приложения

При отправке заказа можно просмотреть данные, сохраненные сервером, перейдя по адресу `http://localhost:4600/orders`, как показано на рис. 17.5.

ПРИМЕЧАНИЕ

Данные о заказах хранятся временно и будут потеряны при остановке или перезапуске веб-сервиса. Возможность постоянного хранения будет добавлена в следующем разделе.



The screenshot shows a browser window with the URL `localhost:4600/orders`. The page content displays a JSON array representing orders:

```
[{"id": 1, "lines": [{"productId": 1, "productName": "Kayak", "quantity": 1}, {"productId": 2, "productName": "Lifejacket", "quantity": 1}], "id": 1}
```

Рис. 17.5. Проверка представленных заказов

17.4. РАЗВЕРТЫВАНИЕ ПРИЛОЖЕНИЯ

Сервер разработки Webpack и инструментарий, предоставляющий ему пакет, не могут быть использованы в версии приложения для конечных пользователей, поэтому для его подготовки к развертыванию требуются некоторые дополнительные шаги, описанные в следующих разделах.

17.4.1. Добавление пакета рабочего варианта HTTP-сервера

Сервер разработки Webpack не следует использовать в реальной эксплуатационной среде, поскольку его функционал ориентирован на динамическое создание пакетов на основе изменений в исходном коде. Для рабочей среды необходим обычный HTTP-сервер для доставки файлов HTML, CSS и JavaScript в браузер, и хорошим выбором для простых проектов является сервер Express с открытым исходным кодом, который представляет собой JavaScript-пакет, исполняемый средой выполнения Node.js. Нажатием **Control+C** остановите работу средств разработки и через консоль в папке `webapp` выполните команду, показанную в листинге 17.14, для установки пакета `express`.

ПРИМЕЧАНИЕ

Пакет `express` может быть уже установлен, поскольку он используется другими инструментами. Тем не менее добавление пакета вручную считается хорошей практикой, так как это создает зависимость в файле `project.json`.

Листинг 17.14. Добавление пакета для развертывания

```
npm install --save-dev express@4.18.2
```

17.4.2. Создание файла для постоянного хранения данных

Пакет json-server будет хранить свои данные постоянно, если он настроен на использование файлов JSON, а не JavaScript, которые позволяют сбрасывать данные в процессе разработки. Добавьте в папку webapp файл с именем `data.json` и поместите в него утверждения из листинга 17.15.

Листинг 17.15. Содержимое файла `data.json` из папки `webapp`

```
{
  "products": [
    { "id": 1, "name": "Kayak", "category": "Watersports",
      "description": "A boat for one person", "price": 275 },
    { "id": 2, "name": "Lifejacket", "category": "Watersports",
      "description": "Protective and fashionable", "price": 48.95 },
    { "id": 3, "name": "Soccer Ball", "category": "Soccer",
      "description": "FIFA-approved size and weight",
      "price": 19.50 },
    { "id": 4, "name": "Corner Flags", "category": "Soccer",
      "description": "Give your playing field a professional touch",
      "price": 34.95 },
    { "id": 5, "name": "Stadium", "category": "Soccer",
      "description": "Flat-packed 35,000-seat stadium",
      "price": 79500 },
    { "id": 6, "name": "Thinking Cap", "category": "Chess",
      "description": "Improve brain efficiency by 75%",
      "price": 16 },
    { "id": 7, "name": "Unsteady Chair", "category": "Chess",
      "description": "Secretly give your opponent a disadvantage",
      "price": 29.95 },
    { "id": 8, "name": "Human Chess Board", "category": "Chess",
      "description": "A fun game for the family", "price": 75 },
    { "id": 9, "name": "Bling Bling King", "category": "Chess",
      "description": "Gold-plated, diamond-studded King",
      "price": 1200 }
  ],
  "orders": []
}
```

Это та же самая информация о товаре, которую мы добавили в файл JavaScript в листинге 17.2, но уже в формате JSON. Теперь данные о заказе не будут потеряны при остановке или перезапуске приложения.

17.4.3. Создание сервера

Для создания сервера, который будет доставлять приложение и его данные в браузер, создайте в папке `webapp` файл с именем `server.js` и поместите в него код, показанный в листинге 17.16.

Листинг 17.16. Содержимое файла `server.js` из папки `webapp`

```
const express = require("express");
const jsonServer = require("json-server");

const app = express();
```

```

app.use("/", express.static("dist"));
app.use("/", express.static("assets"));

const router = jsonServer.router("data.json");
app.use(jsonServer.bodyParser)
app.use("/api", (req, resp, next) => router(req, resp, next));

const port = process.argv[3] || 4000;
app.listen(port, () => console.log('Running on port ${port}'));

```

Утверждения в файле `server.js` настраивают пакеты `express` и `json-server` так, чтобы содержимое папок `dist` и `assets` использовалось для доставки статических файлов, а URL-адреса с префиксом `/api` обрабатывались веб-сервисом.

СОВЕТ

Подобный серверный код можно написать на TypeScript, а затем скомпилировать его, чтобы сгенерировать JavaScript, который будет выполняться в эксплуатируемой среде. Это хорошая идея, если у вас особенно сложный серверный код, но я считаю, что для простых проектов, в которых только комбинируются возможности, предоставляемые различными пакетами, удобнее работать непосредственно на JavaScript.

17.4.4. Использование относительных URL-адресов для запросов данных

Веб-сервис, предоставлявший приложению данные, работает параллельно с сервером разработки Webpack. При развертывании нам необходимо прослушивать оба типа HTTP-запросов на одном порте. Для этого внесите изменения в URL-адреса, используемые классом `RemoteDataSource`, как показано в листинге 17.17.

Листинг 17.17. Использование относительных URL-адресов в файле `remoteDataSource.ts` из папки `src/data`

```

import { AbstractDataSource } from "./abstractDataSource";
import { Product, Order } from "./entities";
import Axios from "axios";

// const protocol = "http";
// const hostname = "localhost";
// const port = 4600;

const urls = {
    // products: `${protocol}://${hostname}:${port}/products`,
    // orders: `${protocol}://${hostname}:${port}/orders`
    products: "/api/products",
    orders: "/api/orders"
};

export class RemoteDataSource extends AbstractDataSource {

    loadProducts(): Promise<Product[]> {
        return Axios.get(urls.products).then(response => response.data);
    }
}

```

```

storeOrder(): Promise<number> {
    let orderData = {
        lines: [...this.order.orderLines.values()].map(ol => ({
            productId: ol.product.id,
            productName: ol.product.name,
            quantity: ol.quantity
        }))
    }
    return Axios.post(urls.orders, orderData)
        .then(response => response.data.id);
}
}
}

```

URL-адреса указываются относительно того URL-адреса, который использовался для запроса HTML-документа, следуя общепринятым соглашениям, согласно которым запросы данных начинаются с префикса /api.

17.4.5. Сборка приложения

В папке webapp выполните команду, показанную в листинге 17.18, для создания пакета, который можно использовать в производственной среде.

Листинг 17.18. Создание производственного пакета

```
npx webpack --mode "production"
```

Если аргумент mode имеет значение production, то webpack создает пакет, содержимое которого минимизировано, то есть оптимизировано по размеру в ущерб читаемости кода. Процесс сборки может занять некоторое время. В результате вы увидите следующее сообщение, где показано, какие файлы были включены в пакет:

```

asset bundle.js 2.23 MiB [emitted] [minimized] [big] (name: main)
orphan modules 99.8 KiB [orphan] 55 modules
runtime modules 1.09 KiB 5 modules
cacheable modules 956 KiB
asset modules 4.4 KiB
  data:image/svg+xml,%3csvg xmlns=%27.. 281 bytes [built] [code generated]
  data:image/svg+xml,%3csvg xmlns=%27.. 279 bytes [built] [code generated]
  data:image/svg+xml,%3csvg xmlns=%27.. 161 bytes [built] [code generated]
  data:image/svg+xml,%3csvg xmlns=%27.. 271 bytes [built] [code generated]
+ 12 modules
javascript modules 952 KiB

modules by path ./node_modules/style-loader/dist/runtime/*.js
  5.84 KiB 6 modules

modules by path ./node_modules/css-loader/dist/runtime/*.js
  3.33 KiB 3 modules
./src/index.ts + 53 modules 99.7 KiB [built] [code generated]

./node_modules/css-
loader/dist/cjs.js!./node_modules/bootstrap/dist/css/bootstrap.css
  843 KiB [built] [code generated]

WARNING in asset size limit: The following asset(s) exceed the recommended
size limit (244 KiB).

```

This can impact web performance.

Assets:

bundle.js (2.23 MiB)

WARNING in entrypoint size limit: The following entrypoint(s) combined asset size exceeds the recommended limit (244 KiB). This can impact web performance.

Entrypoints:

main (2.23 MiB)

bundle.js

WARNING in webpack performance recommendations:

You can limit the size of your bundles by using import() or require.ensure to lazy load some parts of your application.

For more info visit <https://webpack.js.org/guides/code-splitting/>

webpack 5.76.3 compiled with 3 warnings in 4690 microservice

Файлы TypeScript скомпилируются в JavaScript, как это было в режиме разработки, и файл пакета запишется в папку `dist`. Предупреждения о размере созданных файлов можно проигнорировать.

17.4.6. Тестирование производственной сборки

Чтобы убедиться, что процесс сборки прошел успешно и изменения конфигурации вступили в силу, выполните в папке `webapp` команду из листинга 17.19.

Листинг 17.19. Запуск сервера

```
node server.js
```

После чего вы увидите следующий результат:

```
Running on port 4000
```

Откройте веб-браузер и перейдите по адресу `http://localhost:4000`, где будет показано приложение, как на рис. 17.6.

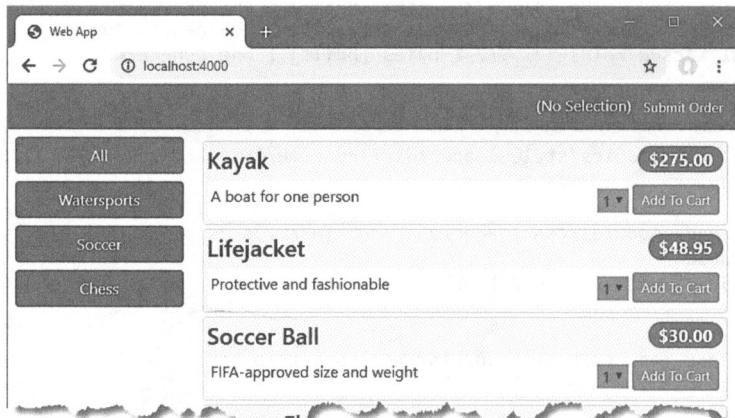


Рис. 17.6. Запуск сборки

17.5. КОНТЕЙНЕРИЗАЦИЯ ПРИЛОЖЕНИЯ

В завершение этой главы мы создадим контейнер для нашего приложения, чтобы его можно было развернуть в производственной среде. На момент написания книги наиболее популярным программным обеспечением для создания контейнера является Docker. Он представляет собой урезанную версию Linux с функциональностью, достаточной для запуска приложения. Большинство облачных платформ или хостинговых движков поддерживают Docker, а его инструменты работают в самых популярных операционных системах.

17.5.1. Установка Docker

Первый шаг – скачать и установить Docker на вашем рабочем компьютере (<https://www.docker.com/products/docker>). Существуют версии для macOS, Windows и Linux, а также несколько специализированных модификаций для работы с облачными платформами Amazon и Microsoft. Для нас достаточно бесплатной версии.

ВНИМАНИЕ

Одним из недостатков Docker является то, что компания, выпускающая данный продукт, приобрела репутацию разработчика, который вносит в свое программное обеспечение критические изменения. Это может означать, что пример, приведенный далее, может работать не так, как предполагалось, в более поздних версиях. Если у вас возникнут проблемы, проверьте репозиторий книги на наличие обновлений (<https://github.com/manningbooks/essential-typescript-5>) или свяжитесь со мной по адресу adam@adam-freeman.com.

17.5.2. Подготовка приложения

Первым делом сформируйте конфигурационный файл для NPM, который будет использоваться для загрузки дополнительных пакетов, необходимых для работы приложения в контейнере. Создайте в папке webapp файл с именем `deploy-package.json` с таким содержимым, как в листинге 17.20.

Листинг 17.20. Содержимое файла `deploy-package.json` из папки `webapp`

```
{  
  "name": "webapp",  
  "description": "Stand-Alone Web App",  
  "repository": "https://github.com/manningbooks/essential-typescript-5",  
  "license": "0BSD",  
  "devDependencies": {  
    "express": "4.18.2",  
    "json-server": "0.17.3"  
  }  
}
```

В разделе `devDependencies` указываются пакеты, необходимые для запуска приложения в контейнере. Все пакеты, для которых в файлах кода приложения имеются операторы `import`, будут включены в пакет, созданный `webpack`, и перечислены в виде списка. Остальные поля описывают приложение и используются преимущественно для предотвращения появления предупреждений при создании контейнера.

17.5.3. Создание контейнера Docker

Чтобы определить контейнер, добавьте в папку `webapp` файл с именем `Dockerfile` (без расширения), содержимое которого показано в листинге 17.21.

Листинг 17.21. Содержимое файла Dockerfile из папки webapp

```
FROM node:18.14.0

RUN mkdir -p /usr/src/webapp

COPY dist /usr/src/webapp/dist
COPY assets /usr/src/webapp/assets

COPY data.json /usr/src/webapp/
COPY server.js /usr/src/webapp/
COPY deploy-package.json /usr/src/webapp/package.json

WORKDIR /usr/src/webapp

RUN echo 'package-lock=false' >> .npmrc
RUN npm install

EXPOSE 4000

CMD ["node", "server.js"]
```

Содержимое файла `Dockerfile` использует базовый образ, сконфигурированный с помощью `Node.js`, и копирует в контейнер файлы, необходимые для запуска приложения, включая файл пакета, содержащий само приложение, а также файл, который будет использоваться для установки пакетов NPM, требуемых для запуска развернутого приложения.

Для того чтобы ускорить процесс контейнеризации, создайте в папке `webapp` файл `.dockerignore` и поместите в него утверждение из листинга 17.22. Теперь Docker будет игнорировать папку `node_modules`, которая не нужна в контейнере и отнимает много времени при обработке.

Листинг 17.22. Содержимое файла .dockerignore из папки webapp

```
node_modules
```

Выполните команду из листинга 17.23 в папке `webapp`, чтобы создать образ приложения вместе со всеми необходимыми пакетами.

Листинг 17.23. Создание образа Docker

```
docker build . -t webapp -f Dockerfile
```

Образ – это шаблон для контейнеров. По мере того как Docker обрабатывает инструкции в файле Docker, загружаются и устанавливаются пакеты NPM, а файлы конфигурации и кода скопируются в образ.

17.5.4. Запуск приложения

После формирования образа создайте и запустите новый контейнер с помощью команды из листинга 17.24.

Листинг 17.24. Запуск контейнера Docker

```
docker run -p 4000:4000 webapp
```

Вы можете протестировать приложение, открыв страницу <http://localhost:4000> в браузере, где будет отображен ответ, выданный веб-сервером, запущенным в контейнере (см. рис. 17.6).

Для вывода списка запущенных контейнеров Docker на текущей машине выполните команду из листинга 17.25.

Листинг 17.25. Вывод списка контейнеров

```
docker ps
```

Вы увидите примерно следующий список (для краткости я опустил некоторые поля):

CONTAINER ID	IMAGE	COMMAND	CREATED
4b9b82772197	webapp	"docker-entrypoint.s..."	33 seconds ago

Используя значение в столбце **CONTAINER ID**, выполните команду, показанную в листинге 17.26, чтобы остановить работу контейнера.

Листинг 17.26. Остановка контейнера

```
docker stop 4b9b82772197
```

Теперь наше приложение готово к развертыванию на любой платформе, поддерживающей Docker.

РЕЗЮМЕ

В этой главе мы завершили разработку автономного веб-приложения, добавив источник данных, использующий веб-сервис, а также JSX-классы, отображающие различный контент для пользователя. Заключительным шагом были подготовка приложения к развертыванию и создание образа контейнера Docker.

- REST-ориентированные веб-сервисы могут быть использованы с помощью стандартных HTTP-запросов, что упрощается при работе с таким пакетом, как Axios.
- Компилятор TypeScript имеет настройку `bundler` для конфигурационного свойства `moduleResolution`, которое можно использовать для поиска модулей в папке `node_modules`, и обеспечивает обработку импорта модулей таким образом, чтобы они взаимодействовали с такими сборщиками, как webpack.
- Приложения TypeScript компилируются в чистый JavaScript, а значит, их можно упаковывать и развертывать так, чтобы они обслуживались из контейнеров.

В следующей главе мы напишем веб-приложение с помощью фреймворка Angular.

18

Создание приложения на Angular, часть 1

В этой главе

- ✓ Создание и настройка проекта Angular.
- ✓ Основы конфигурации Angular TypeScript.
- ✓ Построение модели данных для приложения на Angular.
- ✓ Создание компонентов Angular для базовых функций приложения.
- ✓ Настройка приложения Angular.

В этой главе начинается процесс создания веб-приложения на Angular с тем же набором функций, что и в примерах из глав 16 и 17. В отличие от других фреймворков, где TypeScript является опцией, Angular ставит TypeScript в центр разработки веб-приложений и опирается на его возможности, особенно на декораторы. В качестве краткой справки в табл. 18.1 перечислены параметры компилятора TypeScript, используемые в текущей главе.

Таблица 18.1. Параметры компилятора TypeScript, применяемые в данной главе

Название	Описание
baseUrl	Задает корневое расположение для разрешения зависимостей от модуля
declaration	При включении этого параметра компилятор создает файлы деклараций типов, которые представляют информацию о типах для JavaScript-кода

Название	Описание
downlevelIteration	Включает поддержку итераторов при работе со старыми версиями JavaScript
experimentalDecorators	Включает поддержку декораторов
forceConsistentCasingInFileNames	Гарантирует согласованность регистра имен в операторах import
importHelpers	Определяет, будет ли к JavaScript добавляться вспомогательный код, чтобы уменьшить общий объем кода
lib	Выбирает файлы деклараций типов, которые использует компилятор
module	Задает формат используемых модулей
moduleResolution	Задает стиль разрешения модуля, который следует использовать для разрешения зависимостей
noFallthroughCasesInSwitch	Предотвращает неявные попадания в следующий блок case в switch-конструкциях, если не указан оператор break
noImplicitOverride	Отключает неявное переопределение методов в интерфейсах и классах
noImplicitReturns	Проверяет, чтобы все ветки функции возвращали значение
noPropertyAccessFromIndexSignature	Определяет, разрешено ли обращение к свойствам объекта с использованием точечной нотации, если этот объект имеет сигнатуру индекса
outDir	Задает каталог, в который будут помещены файлы JavaScript
sourceMap	Определяет, генерировать ли компилятору карты исходников для отладки
strict	Позволяет более строго проверять типы, в том числе запретить неявное использование any
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода
useDefineForClassFields	Устанавливает, как определяются поля класса в выводе JavaScript

18.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

Проекты Angular проще всего создавать с помощью пакета `angular-cli`. Откройте командную строку и выполните команду, показанную в листинге 18.1, чтобы установить пакет `angular-cli`.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 18.1. Установка пакета создания проекта

```
npm install --global @angular/cli@15.2.4
```

Имена пакетов Angular начинаются с символа @. После установки пакета перейдите в удобный каталог и выполните команду из листинга 18.2 для создания нового проекта Angular.

Листинг 18.2. Создание нового проекта

```
ng new angularapp
```

Инструменты разработки Angular активируются с помощью команды `ng`, а `ng new` создает новый проект. В процессе установки вам будут предложены варианты настроек нового проекта. Воспользуйтесь ответами из табл. 18.2, чтобы подготовить пример проекта.

Таблица 18.2. Вопросы и ответы по настройке проекта

Вопрос	Ответ
Добавить маршрутизацию Angular?	Да
Какой формат таблицы стилей вы хотите использовать?	CSS

Создание проекта занимает некоторое время из-за большого количества загружаемых пакетов JavaScript.

18.1.1. Настройка веб-сервиса

После завершения процесса создания выполните команды из листинга 18.3, чтобы перейти в папку проекта и добавить пакеты, которые предоставляют веб-сервис, а также позволяют запускать несколько пакетов одной командой.

Листинг 18.3. Добавление пакетов в проект

```
cd angularapp
npm install --save-dev json-server@0.17.3
npm install --save-dev npm-run-all@4.1.5
```

Чтобы предоставить данные для веб-сервиса, поместите в папку `angularapp` файл `data.js` с содержимым, показанным в листинге 18.4.

Листинг 18.4. Содержимое файла data.js из папки angularapp

```
module.exports = function () {
    return {
        products: [
            { id: 1, name: "Kayak", category: "Watersports",
                description: "A boat for one person", price: 275 },
            { id: 2, name: "Lifejacket", category: "Watersports",
                description: "Protective and fashionable", price: 48.95 },
            { id: 3, name: "Soccer Ball", category: "Soccer",
                description: "FIFA-approved size and weight",
                price: 19.50 },
            { id: 4, name: "Corner Flags", category: "Soccer",
                description:
                    "Give your playing field a professional touch",
                price: 34.95 },
            { id: 5, name: "Stadium", category: "Soccer",
                description: "Flat-packed 35,000-seat stadium",
                price: 79500 },
            { id: 6, name: "Thinking Cap", category: "Chess",
                description: "Improve brain efficiency by 75%",
                price: 16 },
            { id: 7, name: "Unsteady Chair", category: "Chess",
                description: "Secretly give your opponent a disadvantage",
                price: 29.95 },
            { id: 8, name: "Human Chess Board", category: "Chess",
                description: "A fun game for the family", price: 75 },
            { id: 9, name: "Bling Bling King", category: "Chess",
                description: "Gold-plated, diamond-studded King",
                price: 1200 }
        ],
        orders: []
    }
}
```

Обновите раздел scripts в файле package.json для настройки средств разработки таким образом, чтобы цепочка инструментов Angular и веб-сервис запускались одновременно, как показано в листинге 18.5.

Листинг 18.5. Настройка инструментов в файле package.json из папки angularapp

```
...
"scripts": {
    "ng": "ng",
    "json": "json-server data.js -p 4600",
    "serve": "ng serve",
    "start": "npm-run-all -p serve json",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
},
...
```

Эти записи позволяют одной командой запустить как веб-сервис, предоставляющий данные, так и инструменты разработки Angular.

18.1.2. Настройка пакета Bootstrap CSS

С помощью командной строки выполните в папке angularapp команду из листинга 18.6 для добавления в проект CSS-фреймворка Bootstrap.

Листинг 18.6. Добавление пакета

```
npm install bootstrap@5.2.3
```

Инструменты разработки Angular требуют изменения конфигурации для включения таблицы стилей CSS Bootstrap в приложение. Откройте файл angular.json в папке angularapp и добавьте в раздел build/styles элемент, показанный в листинге 18.7.

ВНИМАНИЕ

В файле angular.json есть две настройки для стилей, и важно изменить именно ту, что расположена в разделе build, а не test. Если при запуске приложения вы не видите стилизованного содержимого, то, скорее всего, вы отредактировали не тот раздел.

Листинг 18.7. Добавление таблицы стилей в файл angular.json из папки angularapp

```
...
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
    "outputPath": "dist/angularapp",
    "index": "src/index.html",
    "main": "src/main.ts",
    "polyfills": "src/polyfills.ts",
    "tsConfig": "src/tsconfig.app.json",
    "assets": [
      "src/favicon.ico",
      "src/assets"
    ],
    "styles": [
      "src/styles.css",
      "node_modules/bootstrap/dist/css/bootstrap.min.css"
    ],
    "scripts": [],
    "es5BrowserSupport": true
  },
  ...
}
```

18.1.3. Запуск примера приложения

С помощью командной строки выполните в папке angularapp команду, показанную в листинге 18.8.

Листинг 18.8. Запуск средств разработки

```
npm start
```

Инструментам разработки Angular требуется некоторое время, чтобы запуститься и произвести начальную компиляцию, в результате чего получаются следующие результаты:

```
...
✓ Browser application bundle generation complete.

Initial Chunk Files      | Names          | Raw Size
vendor.js                 | vendor         | 2.04 MB |
styles.css, styles.js     | styles         | 398.72 kB |
polyfills.js               | polyfills      | 314.27 kB |
main.js                    | main           | 48.10 kB |
runtime.js                | runtime        | 6.52 kB |

| Initial Total | 2.79 MB
```

Build at: 2023-03-26T07:33:08.269Z - Hash: b52d7ae4c7e8d087 - Time: 3963ms

** Angular Live Development Server is listening on localhost:4200, open your browser on <http://localhost:4200/> **

...

После завершения начальной компиляции откройте окно браузера и перейдите по адресу <http://localhost:4200>, чтобы увидеть содержимое заполнителя, созданное командой из листинга 18.2 и показанное на рис. 18.1.

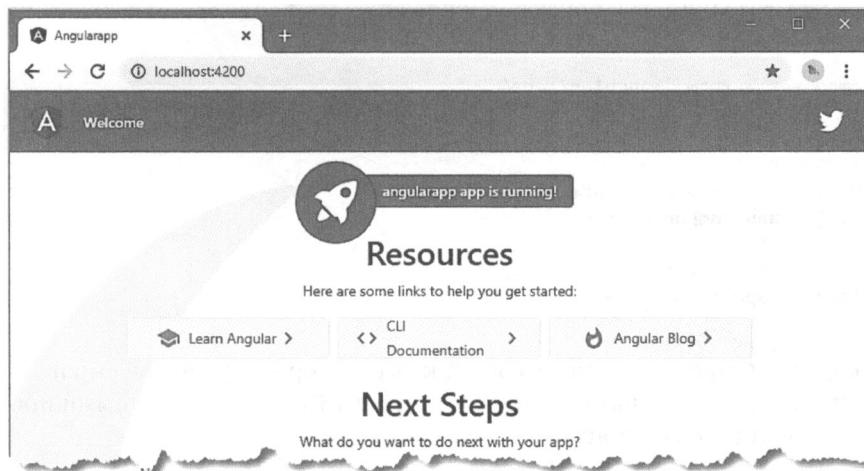


Рис. 18.1. Запуск примера приложения

18.2. РОЛЬ TYPESCRIPT В РАЗРАБОТКЕ ANGULAR

Angular зависит от декораторов TypeScript, но до сих пор не обновлен для использования стандартных декораторов, описанных в главе 14. Вместо этого Angular опирается на предыдущую реализацию декораторов TypeScript, которая в большинстве случаев работает аналогично, но требует некоторых дополнительных настроек компилятора.

Посмотрите на содержимое файла `app.module.ts` в каталоге `src/app`, и вы увидите один из декораторов, от которых зависит Angular.

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, AppRoutingModule],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }

```

Декораторы настолько важны в разработке на Angular, что их применяют к классам, которые содержат мало членов или даже не содержат их вовсе, просто чтобы помочь определить или сконфигурировать приложение. Выше представлен декоратор `NgModule`, и он используется для описания группы связанных функций в приложении Angular (модули Angular существуют наряду с обычными модулями JavaScript, поэтому в этом файле присутствуют и операторы `import`, и декоратор `NgModule`). Другой пример можно увидеть в файле `app.component.ts` в папке `src/app`.

```

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angularapp';
}

```

Это декоратор `Component`, описывающий класс, который будет генерировать HTML-контент, схожий по назначению с классами JSX, ранее созданными при разработке автономного веб-приложения.

18.2.1. Конфигурация компилятора TypeScript

Инструментарий для Angular аналогичен тому, который мы использовали в главах 15 и 16, и основан на `webpack` и сервере разработки `Webpack` с настройками, специфичными для Angular. Следы работы `webpack` заметны в некоторых сообщениях, выдаваемых инструментами разработки Angular, но подробности и конфигурационный файл напрямую не раскрываются. Конфигурацию, используемую для компилятора TypeScript, можно увидеть и изменить, поскольку проект создается с файлом `tsconfig.json`, в котором указаны следующие настройки:

```
{
  "compileOnSave": false,
  "compilerOptions": {
    ...
  }
}
```

```
"baseUrl": "./",
"outDir": "./dist/out-tsc",
"forceConsistentCasingInFileNames": true,
"strict": true,
"noImplicitOverride": true,
"noPropertyAccessFromIndexSignature": true,
"noImplicitReturns": true,
"noFallthroughCasesInSwitch": true,
"sourceMap": true,
"declaration": false,
"downlevelIteration": true,
"experimentalDecorators": true,
"moduleResolution": "node",
"importHelpers": true,
"target": "ES2022",
"module": "ES2022",
"useDefineForClassFields": false,
"lib": [
  "ES2022",
  "dom"
]
},
"angularCompilerOptions": {
  "enableI18nLegacyMessageIdFormat": false,
  "strictInjectionParameters": true,
  "strictInputAccessModifiers": true,
  "strictTemplates": true
}
}
```

Конфигурация записывает скомпилированные JavaScript-файлы в каталог `dist/out-tsc`, хотя в проекте он не виден, поскольку webpack используется для автоматического формирования пакета.

Наиболее важной настройкой является `experimentalDecorators`, которая позволяет реализовать декораторы, требуемые фреймворком Angular.

ВНИМАНИЕ

При внесении изменений в файл `tsconfig.json` необходимо соблюдать осторожность, поскольку различные модификации могут привести к поломке остальной части цепочки инструментов Angular. Большинство изменений в проекте Angular вносятся через файл `angular.json`.

18.3. СОЗДАНИЕ МОДЕЛИ ДАННЫХ

Для запуска модели данных создайте папку `src/app/data` и добавьте в нее файл с именем `entities.ts` с кодом, показанным в листинге 18.9.

Листинг 18.9. Содержимое файла `entities.ts` из папки `src/app/data`

```
export type Product = {
  id: number,
  name: string,
  description: string,
```

```

        category: string,
        price: number
    };

    export class OrderLine {
        constructor(public product: Product, public quantity: number) {
            // не требуется никаких операторов
        }

        get total(): number {
            return this.product.price * this.quantity;
        }
    }

    export class Order {
        private lines = new Map<number, OrderLine>();

        constructor(initialLines?: OrderLine[]) {
            if (initialLines) {
                initialLines.forEach(ol => this.lines.set(ol.product.id, ol));
            }
        }

        public addProduct(prod: Product, quantity: number) {
            if (this.lines.has(prod.id)) {
                if (quantity === 0) {
                    this.removeProduct(prod.id);
                } else {
                    this.lines.get(prod.id)!.quantity += quantity;
                }
            } else {
                this.lines.set(prod.id, new OrderLine(prod, quantity));
            }
        }

        public removeProduct(id: number) {
            this.lines.delete(id);
        }

        get orderLines(): OrderLine[] {
            return [...this.lines.values()];
        }

        get productCount(): number {
            return [...this.lines.values()]
                .reduce((total, ol) => total += ol.quantity, 0);
        }

        get total(): number {
            return [...this.lines.values()]
                .reduce((total, ol) => total += ol.total, 0);
        }
    }
}

```

Это код из главы 15, и он не требует изменений, поскольку Angular использует обычные классы TypeScript для сущностей своей модели данных.

18.3.1. Создание источника данных

Чтобы создать источник данных, добавьте в папку `src/app/data` файл с именем `dataSource.ts` с кодом, показанным в листинге 18.10.

Листинг 18.10. Содержимое файла `dataSource.ts` из папки `src/app/data`

```
import { Observable } from "rxjs";
import { Injectable } from '@angular/core';
import { Product, Order } from "./entities";

export type ProductProp = keyof Product;

export abstract class DataSourceImpl {
    abstract loadProducts(): Observable<Product[]>;
    abstract storeOrder(order: Order): Observable<number>;
}

@Injectable()
export class DataSource {
    private _products: Product[];
    private _categories: Set<string>;
    public order: Order;

    constructor(private impl: DataSourceImpl) {
        this._products = [];
        this._categories = new Set<string>();
        this.order = new Order();
        this.getData();
    }

    getProducts(sortProp: ProductProp = "id", category? : string)
        : Product[] {
        return this.selectProducts(this._products, sortProp, category);
    }

    protected getData(): void {
        this._products = [];
        this._categories.clear();
        this.impl.loadProducts().subscribe(rawData => {
            rawData.forEach(p => {
                this._products.push(p);
                this._categories.add(p.category);
            });
        });
    }

    protected selectProducts(prods: Product[], sortProp: ProductProp,
        category?: string): Product[] {
        return prods
            .filter(p => category === undefined || p.category === category)
            .sort((p1, p2) => p1[sortProp] < p2[sortProp]
                ? -1 : p1[sortProp] > p2[sortProp] ? 1: 0);
    }
}
```

```

    getCategories(): string[] {
      return [...this._categories.values()];
    }

    storeOrder(): Observable<number> {
      return this.impl.storeOrder(this.order);
    }
}

```

Сервисы — одна из ключевых особенностей разработки на Angular. Они позволяют классам объявлять в своих конструкторах зависимости, которые разрешаются во время выполнения программы, используя технику, известную как *внедрение зависимостей*. Класс `DataSource` объявляет в своем конструкторе зависимость от объекта `DataSourceImpl`, например, так:

```

...
constructor(private impl: DataSourceImpl) {
...

```

Когда возникает необходимость в новом объекте `DataSource`, Angular проверяет конструктор, создает объект `DataSourceImpl` и использует его для вызова конструктора для создания нового объекта, что называется *внедрением*. Декоратор `Injectable` сообщает Angular, что другие классы могут объявлять зависимости от класса `DataSource`. Класс `DataSourceImpl` является абстрактным, и класс `DataSource` не имеет представления о том, какой конкретный класс реализации будет использован для разрешения зависимости от его конструктора. Выбор класса реализации осуществляется в конфигурации приложения, как показано в листинге 18.12 далее.

Одним из ключевых преимуществ фреймворка в разработке веб-приложений является автоматическое управление обновлениями. Для этого в Angular используется библиотека Reactive Extensions, известная как RxJS, которая позволяет автоматически обрабатывать изменения данных. Класс `Observable` из RxJS используется для описания последовательности значений, которые будут генерироваться с течением времени, включая асинхронные операции, такие как запрос данных из веб-сервиса. Метод `loadProducts`, определенный классом `DataSourceImpl`, возвращает объект `Observable<Product[]>`, например, так:

```

...
abstract loadProducts(): Observable<Product[]>;
...

```

Аргумент обобщенного типа TypeScript используется для указания того, что результатом работы метода `loadProducts` является объект `Observable`, который будет генерировать последовательность объектов массива `Product`. Значения, сгенерированные объектом `Observable`, принимаются с помощью метода `subscribe`:

```

...
this.impl.loadProducts().subscribe(rawData => {
  rawData.forEach(p => {
    this._products.push(p);
    this._categories.add(p.category);
  });
});
...

```

В данной ситуации мы используем класс `Observable` в качестве прямой замены стандартного JavaScript `Promise`. Класс `Observable` предоставляет современные функции для работы со сложными последовательностями, но в нашем случае преимущество заключается в том, что Angular будет обновлять содержимое, представляемое пользователю, когда `Observable` выдает результат, что позволяет оставальной части класса `DataSource` быть написанной без необходимости решать асинхронные задачи.

18.3.2. Создание класса реализации источника данных

Чтобы расширить абстрактный класс `DataSourceImpl` для работы с веб-сервисом, добавим в папку `src/app/data` файл с именем `remoteDataSource.ts` и поместим в него код из листинга 18.11.

Листинг 18.11. Содержимое файла `remoteDataSource.ts` из папки `src/app/data`

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { map } from "rxjs/operators";
import { DataSourceImpl } from "./dataSource";
import { Product, Order } from "./entities";

const protocol = "http";
const hostname = "localhost";
const port = 4600;

const urls = {
  products: `${protocol}://${hostname}:${port}/products`,
  orders: `${protocol}://${hostname}:${port}/orders`
};

@Injectable()
export class RemoteDataSource extends DataSourceImpl {

  constructor(private http: HttpClient) {
    super();
  }

  loadProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(urls.products);
  }

  storeOrder(order: Order): Observable<number> {
    let orderData = {
      lines: [...order.orderLines.values()].map(ol => ({
        productId: ol.product.id,
        productName: ol.product.name,
        quantity: ol.quantity
      }))
    }
    return this.http.post<{ id: number }>(urls.orders, orderData)
      .pipe<number>(map(val => val.id));
  }
}
```

Конструктор `RemoteDataSource` объявляет зависимость от экземпляра класса `HttpClient`, который является встроенным классом Angular для выполнения HTTP-запросов. Класс `HttpClient` определяет методы `get` и `post`, необходимые для отправки HTTP-запросов с методами GET и POST. В качестве аргумента типа указывается ожидаемый тип данных:

```
...
loadProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(urls.products);
}
...
```

Аргумент типа используется для результата метода `get`, представляющего собой объект `Observable`, который генерирует последовательность указанного типа, в данном случае `Product[]`.

СОВЕТ

Аргументы обобщенного типа для методов `HttpClient` являются стандартными для TypeScript. Никакой магии Angular не происходит, и разработчик остается ответственным за указание типа, который будет соответствовать данным, полученным от сервера.

Библиотека RxJS содержит функции, позволяющие манипулировать значениями, генерируемыми объектом `Observable`, часть из которых используется в листинге 18.11.

```
...
return this.http.post<{ id: number}>(urls.orders, orderData)
    .pipe<number>(map(val => val.id));
...
```

Метод `pipe` используется с функцией `map` для создания `Observable`, который генерирует значения на основе значений из другого `Observable`. Это позволяет нам получить результат HTTP-запроса POST и извлечь из него только свойство `id`.

ПРИМЕЧАНИЕ

В автономном веб-приложении я создал абстрактный класс источника данных и подклассы, предоставляющие локальные данные или данные веб-сервиса, которые были загружены с помощью метода, вызванного в конструкторе абстрактного класса. Такой подход не очень хорошо работает в Angular, поскольку `HttpClient` присваивается свойству экземпляра только после вызова конструктора абстрактного класса с ключевым словом `super`. Это означает, что подклассу предлагается получать данные до того, как он будет правильно настроен. Чтобы избежать подобной проблемы, я выделил в абстрактный класс только ту часть источника данных, которая работает с данными.

18.3.3. Настройка источника данных

Последним шагом формирования источника данных является создание модуля Angular, который сделает источник данных доступным для использования в оставшейся части приложения и выберет реализацию абстрактного класса `DataSourceImpl`. Добавьте в папку `src/app/data` файл с именем `data.module.ts` и поместите в него код, показанный в листинге 18.12.

Листинг 18.12. Содержимое файла data.module.ts из папки src/app/data

```
import { NgModule } from "@angular/core";
import { HttpClientModule } from "@angular/common/http";
import { DataSource, DataSourceImpl } from './dataSource';
import { RemoteDataSource } from './remoteDataSource';

@NgModule({
  imports: [HttpClientModule],
  providers: [DataSource,
    { provide: DataSourceImpl, useClass: RemoteDataSource }]
})
export class DataModelModule { }
```

Класс `DataModelModule` определяется исключительно для применения декоратора `NgModule`. Свойство `imports` декоратора определяет зависимости, требуемые классам модели данных, а свойство `providers` – классы в модуле Angular, которые можно внедрять в конструкторы других классов в приложении. Для этого модуля свойство `imports` сообщает Angular, что требуется модуль, содержащий класс `HttpClient`, а свойство `providers` указывает Angular, что класс `DataSource` может быть использован для внедрения зависимостей и что зависимости от класса `DataSourceImpl` следует разрешать с помощью класса `RemoteDataSource`.

18.4. ОТОБРАЖЕНИЕ ОТФИЛЬТРОВАННОГО СПИСКА ТОВАРОВ

Angular разделяет генерацию HTML-контента на два файла: класс TypeScript, к которому используется декоратор `Component`, и HTML-шаблон, аннотированный директивами, указывающими на генерацию динамического контента. При выполнении приложения HTML-шаблон компилируется и директивы выполняются с использованием методов и свойств, предоставляемых классом TypeScript.

Классы, к которым используется декоратор `Component`, называются *компонентами*. При разработке на Angular принято включать роль класса в имя файла. Так, для создания компонента, отвечающего за предоставление пользователю подробной информации об отдельном товаре, мы добавим в папку `src/app` файл `productItem.component.ts` с кодом, показанным в листинге 18.13.

Листинг 18.13. Содержимое файла productItem.component.ts из папки src/app

```
import { Component, Input, Output, EventEmitter } from "@angular/core";
import { Product } from './data/entities';

export type productSelection = {
  product: Product,
  quantity: number
}

@Component({
  selector: "product-item",
  templateUrl: "./productItem.component.html"
})
```

```

export class ProductItem {
    quantity: number = 1;

    @Input()
    product: Product = {
        id: 0, name: "", description: "", category: "", price: 0
    }

    @Output()
    addToCart = new EventEmitter<productSelection>();

    handleAddToCart() {
        this.addToCart.emit({ product: this.product,
            quantity: Number(this.quantity) });
    }
}

```

Декоратор `Component` настраивает компонент. Свойство `selector` указывает CSS-селектор, который Angular использует для применения компонента к HTML-разметке приложения, а свойство `templateUrl` определяет HTML-шаблон компонента. Для класса `ProductItem` свойство `selector` указывает Angular применять этот компонент при обнаружении элемента `product-item`, а HTML-шаблон компонента можно найти в файле с именем `productItem.component.html` в той же директории, что и файл TypeScript.

Angular использует декоратор `Input` для обозначения свойств, позволяющих компонентам получать значения данных через атрибуты HTML-элементов. Декоратор `Output` используется для обозначения потока данных, выводимых из компонента через пользовательское событие. Класс `ProductItem` получает объект `Product`, сведения о котором он отображает пользователю, и запускает пользовательское событие при нажатии пользователем кнопки, доступной через свойство `addToCart`.

Для формирования шаблона компонента создайте в папке `src/app` файл `productItem.component.html` и поместите в него элементы, показанные в листинге 18.14.

Листинг 18.14. Содержимое файла `productItem.component.html` из папки `src/app`

```

<div class="card m-1 p-1 bg-light">
    <h4>
        {{ product.name }}
        <span class="badge rounded-pill bg-primary float-end">
            ${{ product.price.toFixed(2) }}
        </span>
    </h4>
    <div class="card-text bg-white p-1">
        {{ product.description }}
        <button class="btn btn-success btn-sm float-end"
            (click)="handleAddToCart()">
            Add To Cart
        </button>
        <select class="form-control-inline float-end m-1"
            [(ngModel)]="quantity">
            <option>1</option>
            <option>2</option>
        </select>
    </div>
</div>

```

```
<option>3</option>
</select>
</div>
</div>
```

В шаблонах Angular используются двойные фигурные скобки для отображения результатов выражений JavaScript, таких как это:

```
...
<span class="badge rounded-pill bg-primary float-end">
  ${{ product.price.toFixed(2) }}
</span>
...
```

Выражения оцениваются в контексте компонента, поэтому в данном фрагменте считывается значение свойства `product.price`, вызывается метод `toFixed`, а результат вставляется в замыкающий элемент `span`.

Обработка событий осуществляется с помощью круглых скобок вокруг имени события:

```
...
<button class="btn btn-success btn-sm float-end"
        (click)="handleAddToCart()">
...
```

Это указывает Angular, что когда элемент `button` выдает событие `click`, следует вызвать метод `handleAddToCart` компонента. Элементы форм имеют в Angular специальную поддержку, которую видно на примере элемента `select`.

```
...
<select class="form-control-inline float-end m-1" [(ngModel)]="quantity">
...
```

Непосредственно `ngModel` используется с квадратными и круглыми скобками, создавая двустороннюю привязку между элементом `select` и свойством `quantity` компонента. Изменения свойства `quantity` будут отражаться в элементе `select`, а значения, выбранные с помощью элемента `select`, будут использоваться для обновления свойства `quantity`.

18.4.1. Отображение кнопок категорий

Для создания компонента, который будет отображать список кнопок категорий, добавьте в папку `src/app` файл `categoryList.component.ts` и поместите в него код из листинга 18.15.

Листинг 18.15. Содержимое файла `categoryList.component.ts` из папки `src/app`

```
import { Component, Input, Output, EventEmitter } from "@angular/core";

@Component({
  selector: "category-list",
  templateUrl: "./categoryList.component.html"
})
```

```
export class CategoryList {
    @Input()
    selected: string = ""

    @Input()
    categories: string[] = [];

    @Output()
    selectCategory = new EventEmitter<string>();

    getBtnClass(category: string): string {
        return "btn btn-block " +
            (category === this.selected ? "btn-primary" : "btn-secondary");
    }
}
```

Компонент `CategoryList` имеет свойства `Input`, которые получают текущую выбранную категорию и список категорий для отображения. Декоратор `Output` используется к свойству `selectCategory` для создания пользовательского события, которое будет срабатывать, когда пользователь сделает выбор. Метод `getBtnClass` является вспомогательным и возвращает список классов Bootstrap, которым должен быть присвоен элемент `button`, что помогает избавить шаблон компонента от сложных выражений. Для формирования шаблона компонента создайте в папке `src/app` файл с именем `categoryList.component.html` и поместите в него содержимое из листинга 18.16.

Листинг 18.16. Содержимое файла categoryList.component.html из папки src/app

```
<div class="d-grid gap-2">
    <button *ngFor="let cat of categories" [class]="getBtnClass(cat)"
        (click)="selectCategory.emit(cat)">
        {{ cat }}
    </button>
</div>
```

Данный шаблон использует директиву `ngFor` для создания элемента `button` для каждого из значений, возвращаемых свойством `categories`. Символ `*` в префиксе `ngFor`, указывает на лаконичный синтаксис, который позволяет применить директиву `ngFor` непосредственно к генерируемому элементу.

В шаблонах Angular квадратные скобки используются для создания односторонней привязки между атрибутом и значением данных:

```
...
<button *ngFor="let cat of categories" [class]="getBtnClass(cat)"
    (click)="selectCategory.emit(cat)">
...
...
```

Квадратные скобки позволяют задать значение атрибута `class` с помощью выражения JavaScript, которое представляет собой результат вызова метода `getBtnClass` компонента.

18.4.2. Создание отображения заголовка

Чтобы создать компонент, который будет отображать сводку выбранных пользователем товаров и предоставлять средства навигации по ней, добавьте в папку src/app файл с именем header.component.ts с кодом из листинга 18.17.

Листинг 18.17. Содержимое файла header.component.ts из папки src/app

```
import { Component, Input, Output, EventEmitter } from "@angular/core";
import { Order } from './data/entities';

@Component({
  selector: "header",
  templateUrl: "./header.component.html"
})
export class Header {

  @Input()
  order = new Order();

  @Output()
  submit = new EventEmitter<void>();

  get headerText(): string {
    let count = this.order.productCount;
    return count === 0 ? "(No Selection)"
      : `${ count } product(s), ${ this.order.total.toFixed(2) }`
  }
}
```

Для создания шаблона компонента добавьте в папку src/app файл header.component.html, содержимое которого показано в листинге 18.18.

Листинг 18.18. Содержимое файла header.component.html из папки src/app

```
<div class="p-1 bg-secondary text-white text-end">
  {{ headerText }}
  <button class="btn btn-sm btn-primary m-1" (click)="submit.emit()">
    Submit Order
  </button>
</div>
```

18.4.3. Объединение компонентов

Чтобы определить компонент, который представляет пользователю компоненты ProductItem, CategoryList и Header, добавьте в папку src/app файл productList.component.ts с кодом из листинга 18.19.

Листинг 18.19. Содержимое файла productList.component.ts из папки src/app

```
import { Component } from "@angular/core";
import { DataSource } from './data(dataSource';
import { Product } from './data/entities';
```

```

@Component({
  selector: "product-list",
  templateUrl: "./productList.component.html"
})
export class ProductList {
  selectedCategory = "All";

  constructor(public dataSource: DataSource) {}

  get products(): Product[] {
    return this.dataSource.getProducts("id",
      this.selectedCategory === "All"
        ? undefined : this.selectedCategory);
  }

  get categories(): string[] {
    return ["All", ...this.dataSource.getCategories()];
  }

  handleCategorySelect(category: string) {
    this.selectedCategory = category;
  }

  handleAdd(data: {product: Product, quantity: number}) {
    this.dataSource.order.addProduct(data.product, data.quantity);
  }

  handleSubmit() {
    console.log("SUBMIT");
  }
}

```

Класс `ProductList` объявляет зависимость от класса `DataSource` и определяет методы `products` и `categories`, возвращающие данные из источника `DataSource`. Три метода реагируют на взаимодействие с пользователем: `handleCategorySelect` вызывается, когда пользователь нажимает на кнопку категории, `handleAdd` — когда пользователь добавляет товар в заказ, а `handleSubmit` — когда пользователь хочет перейти к сводке заказа. Метод `handleSubmit` выводит сообщение в консоль и будет полностью реализован в текущей главе.

Для создания шаблона компонента добавьте в папку `src/app` файл с именем `productList.component.html`, содержимое которого показано в листинге 18.20.

Листинг 18.20. Содержимое файла `productList.component.html` из папки `src/app`

```

<header [order]="dataSource.order" (submit)="handleSubmit()"></header>
<div class="container-fluid">
  <div class="row">
    <div class="col-3 p-2">
      <category-list [selected]="selectedCategory"
        [categories]="categories"
        (selectCategory)="handleCategorySelect($event)">
      </category-list>
    </div>
    <div class="col-9 p-2">
      <product-item *ngFor="let p of products" [product]="p"
        (addToCart)="handleAdd($event)"></product-item>
    </div>
  </div>
</div>

```

```
        </div>
    </div>

</div>
```

Этот шаблон демонстрирует, как комбинируются компоненты для представления контента пользователю. Пользовательские HTML-элементы, теги которых соответствуют свойствам `selector` в декораторах `Component`, применяются к классам, определенным в предыдущих листингах, следующим образом:

```
...
<header [order]="dataSource.order" (submit)="handleSubmit()"></header>
...
```

Тег `header` соответствует настройке `selector` декоратора `Component`, примененного к классу `Header` в листинге 18.17. Атрибут `order` используется для задания значения свойства `Input` с тем же именем, определенного классом `Header`, и дает возможность `ProductList` предоставить `Header` необходимые данные. Атрибут `submit` соответствует одноименному свойству в `Output`, определенному классом `Header`, и позволяет `ProductList` получать уведомления. Шаблон `ProductList` использует элементы `header`, `category-list` и `product-item` для отображения компонентов `Header`, `CategoryList` и `ProductItem`.

18.5. НАСТРОЙКА ПРИЛОЖЕНИЯ

Модуль приложения применяется для регистрации компонентов, используемых приложением, а также любых дополнительных модулей, которые были определены, например модуля, созданного для модели данных ранее в текущей главе. В листинге 18.21 показаны изменения в модуле приложения, который определен в файле `app.module.ts`.

Листинг 18.21. Настройка модуля в файле `app.module.ts` из папки `src/app`

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';
import { DataModelModule } from "./data/data.module";
import { ProductItem } from './productItem.component';
import { CategoryList } from './categoryList.component';
import { Header } from './header.component';
import { ProductList } from './productList.component';

@NgModule({
  declarations: [AppComponent, ProductItem, CategoryList,
    Header, ProductList],
  imports: [BrowserModule, AppRoutingModule, FormsModule, DataModelModule],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

Свойство `declarations` декоратора `NgModule` нужно для объявления компонентов, необходимых приложению, и для добавления классов, определенных в предыдущих разделах. Свойство `imports` используется для перечисления других модулей, необходимых приложению, и было обновлено для включения модуля модели данных, определенного в листинге 18.12.

Чтобы отобразить новые компоненты пользователю, замените содержимое файла `app.component.html` выражением, показанным в листинге 18.22.

Листинг 18.22. Замена файла `app.component.html` из папки `src/app`

```
<product-list></product-list>
```

При запуске приложения Angular встречает элемент `product-list` и сравнивает его со свойствами `selector` декораторов `Component`, настроенных через модуль Angular. Тег `product-list` соответствует свойству `selector` декоратора `Component`, примененного к классу `ProductList` в листинге 18.19. Angular создает новый объект `ProductList`, отображает содержимое его шаблона и вставляет его в элемент `product-list`, определенный в листинге 18.22. Генерируемый компонентом `ProductList` HTML-код проверяется, обнаруживаются элементы `header`, `category-list` и `product-item`, что приводит к созданию экземпляров этих компонентов и вставке их содержимого в каждый элемент. Процесс повторяется до тех пор, пока все элементы, соответствующие компонентам, не будут разрешены и содержимое не будет представлено пользователю, как показано на рис. 18.2.

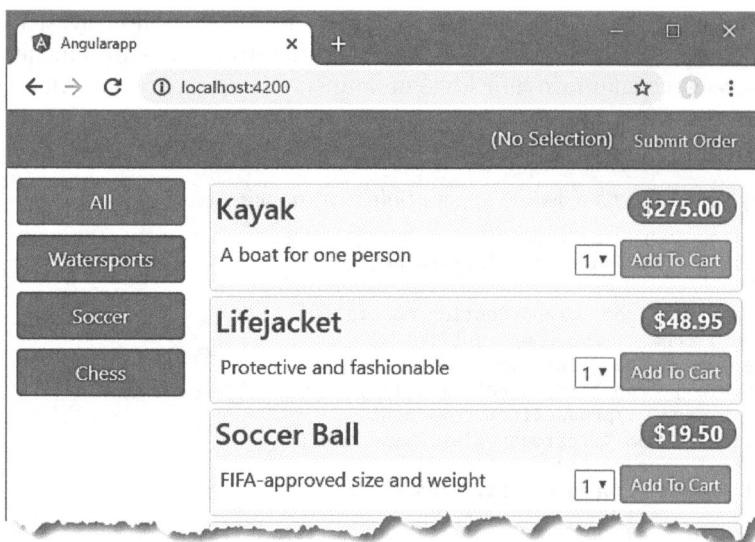


Рис. 18.2. Отображение контента пользователю

Пользователь может фильтровать список товаров и добавлять товары в заказ. При нажатии кнопки `Submit Order` (Отправить заказ) в консоли браузера лишь отобразится сообщение, но в следующей главе мы расширим функционал приложения.

РЕЗЮМЕ

В этой главе вы узнали о роли TypeScript в разработке приложений на Angular. Познакомились с декораторами TypeScript, которые применяются для описания различных строительных блоков. Было также объяснено, что HTML-шаблоны Angular компилируются при выполнении приложения браузером, а это означает, что функции TypeScript уже удалены и не могут использоваться в шаблонах.

- Angular предоставляет инструменты для создания проектов, включая настройку компилятора TypeScript.
- Версия TypeScript, используемая при разработке на Angular, определяется версией самого Angular.
- Angular использует декораторы, но пока не поддерживает новейшие возможности декораторов, описанные в главе 14.
- Декораторы Angular применяются для связывания шаблонов с кодом, позволяя определять сложную разметку отдельно от кода, который предоставляет необходимые данные.

В следующей главе мы завершим работу над приложением и подготовим его к развертыванию.

Создание приложения на Angular, часть 2

В этой главе

- ✓ Добавление поддержки маршрутизации URL в Angular.
- ✓ Создание сервера развертывания и постоянного хранилища данных.
- ✓ Развертывание приложения в контейнере.

В этой главе продолжается разработка веб-приложения Angular, начатая в предыдущей. В рамках данного процесса мы добавим оставшиеся функции и подготовим приложение к развертыванию в контейнере. В качестве краткой справки в табл. 19.1 перечислены параметры компилятора TypeScript, используемые в текущей главе.

Таблица 19.1. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
baseUrl	Задает корневое расположение для разрешения зависимостей от модуля
declaration	При включении этого параметра компилятор создает файлы деклараций типов, которые предоставляют информацию о типах для JavaScript-кода
downlevelIteration	Включает поддержку итераторов при работе со старыми версиями JavaScript
experimentalDecorators	Включает поддержку декораторов

Название	Описание
forceConsistentCasingInFileNames	Гарантирует согласованность регистра имен в операторах <code>import</code>
importHelpers	Определяет, будет ли к JavaScript добавляться вспомогательный код, чтобы уменьшить общий объем кода
lib	Выбирает файлы деклараций типов, которые использует компилятор
module	Задает формат используемых модулей
moduleResolution	Задает стиль разрешения модуля, который следует использовать для разрешения зависимостей
noFallthroughCasesInSwitch	Предотвращает неявные попадания в следующий блок <code>case</code> в <code>switch</code> -конструкциях, если не указан оператор <code>break</code>
noImplicitOverride	Включает выдачу ошибки при неявном переопределении методов в интерфейсах и классах, определенных базовым классом, без использования ключевого слова <code>override</code>
noImplicitReturns	Включает выдачу ошибки для функций и методов, возвращающих необъявленные результаты
noPropertyAccessFromIndexSignature	Определяет, разрешено ли обращение к свойствам объекта, используя точечную нотацию, если этот объект имеет сигнатуру индекса
outDir	Задает каталог, в который будут помещены файлы JavaScript
sourceMap	Определяет, генерировать ли компилятору карты исходников для отладки
strict	Включает более строгую проверку типов, в том числе запрет неявного использования <code>any</code>
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода
useDefineForClassFields	Устанавливает, как определяются поля класса в выводе JavaScript

19.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В текущей главе мы продолжаем работать с проектом `angularapp`, начатым в главе 18. Никаких изменений для того, чтобы начать работу с материалом из этой главы, не требуется. Откройте новое окно командной строки, перейдите в папку `angularapp` и выполните команду, показанную в листинге 19.1, чтобы запустить веб-сервис и инструменты разработки Angular.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 19.1. Запуск средств разработки

```
npm start
```

После завершения сборки откройте новое окно браузера и перейдите по адресу <http://localhost:4200>. Вы увидите пример приложения, как на рис. 19.1.

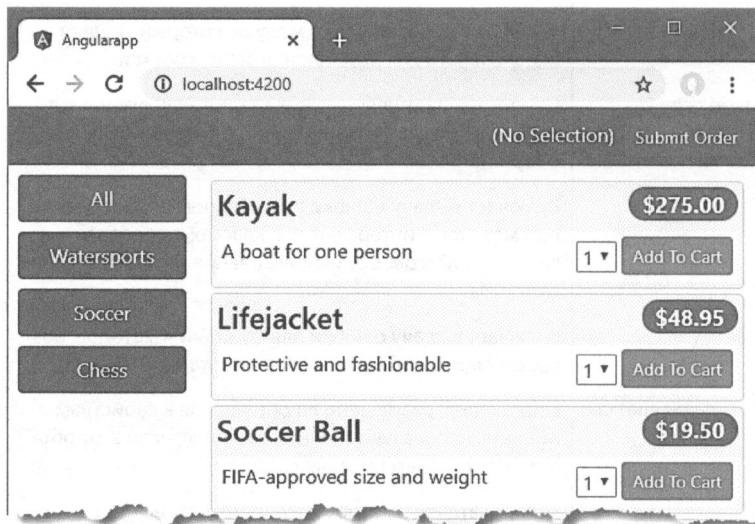


Рис. 19.1. Запуск примера приложения

19.2. ЗАВЕРШЕНИЕ РАБОТЫ НАД ПРИЛОЖЕНИЕМ

Для компонента, который будет отображать детали заказа, добавьте в папку `src/app` файл с именем `orderDetails.component.ts` с кодом из листинга 19.2.

Листинг 19.2. Содержимое файла `orderDetails.component.ts` из папки `src/app`

```
import { Component } from "@angular/core";
import { Router } from "@angular/router";
import { Order } from "./data/entities";
import { DataSource } from './data(dataSource';

@Component({
  selector: "order-details",
  templateUrl: "./orderDetails.component.html"
})
export class OrderDetails {
```

```

constructor(private dataSource: DataSource, private router: Router) {}

get order() : Order {
    return this.dataSource.order;
}

submit() {
    this.dataSource.storeOrder().subscribe(id =>
        this.router.navigateByUrl('/summary/${id}'));
}
}

```

Компонент `OrderDetails` получает объект `DataSource` через свой конструктор и предоставляет свойство `order` своему шаблону. Этот компонент использует систему маршрутизации Angular URL, которая выбирает отображаемые пользователю компоненты в зависимости от текущего URL. В табл. 19.2 приведены поддерживаемые приложением URL-адреса и их назначение.

Таблица 19.2. URL-адреса, поддерживаемые приложением

Название	Описание
/products	Отобразит компонент <code>ProductList</code> , определенный в главе 18
/order	Отобразит компонент <code>OrderDetails</code> , определенный в листинге 19.2
/summary	Отобразит сводную информацию о заказе после его отправки на сервер. В URL будет включен номер заказа. Например, если ID заказа — 5, то URL будет /summary/5
/	По умолчанию этот URL будет перенаправлен на /products, что позволит отобразить компонент <code>ProductList</code>

Объект `Router`, полученный в конструкторе `OrderDetails`, позволяет компоненту использовать функцию маршрутизации URL для перехода на новый URL и используется в методе `submit`.

```

...
submit() {
    this.dataSource.storeOrder().subscribe(id =>
        this.router.navigateByUrl('/summary/${id}'));
}
...

```

Этот метод использует источник данных `DataSource` для отправки заказа пользователя на сервер, ожидает ответа, а затем задействует метод `navigateByUrl` объекта `Router` для перехода к URL, который отобразит пользователю сводку.

Чтобы создать шаблон компонента `OrderDetails`, добавьте в папку `src/app` файл с именем `orderDetails.component.html`, содержимое которого показано в листинге 19.3.

Листинг 19.3. Содержимое файла orderDetails.component.html из папки src/app

```

<h3 class="text-center bg-primary text-white p-2">Order Summary</h3>
<div class="p-3">
  <table class="table table-sm table-striped">
    <thead>
      <tr>
        <th>Quantity</th><th>Product</th>
        <th class="text-end">Price</th>
        <th class="text-end">Subtotal</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let line of order.orderLines">
        <td>{{ line.quantity }}</td>
        <td>{{ line.product.name }}</td>
        <td class="text-end">
          ${{ line.product.price.toFixed(2) }}
        </td>
        <td class="text-end">${{ line.total.toFixed(2) }}</td>
      </tr>
    </tbody>
    <tfoot>
      <tr>
        <th class="text-end" colSpan="3">Total:</th>
        <th class="text-end">
          ${{ order.total.toFixed(2) }}
        </th>
      </tr>
    </tfoot>
  </table>
</div>
<div class="text-center">
  <button class="btn btn-secondary m-1" routerLink="/products">
    Back
  </button>
  <button class="btn btn-primary m-1" (click)="submit()">
    Submit Order
  </button>
</div>

```

Компонент выводит подробную информацию о выбранных пользователем товарах, а также кнопки, вызывающие метод `submit` или осуществляющие переход к списку `/products`, чтобы отобразить компонент `ProductList`. Навигация настраивается путем применения директивы `routerLink` к элементу `button` и указания URL, на который браузер перейдет при нажатии на элемент.

```

...
<button class="btn btn-secondary m-1" routerLink="/products">Back</button>
...

```

Директива `routerLink` является частью функционала маршрутизации в Angular и позволяет осуществлять навигацию без необходимости использования объекта `Router` в классе компонента.

19.2.1. Добавление компонента сводки

Для создания компонента, который будет отображаться для URL /summary, добавьте в папку src/app файл с именем `summary.component.ts` с кодом из листинга 19.4.

Листинг 19.4. Содержимое файла `summary.component.ts` из папки `src/app`

```
import { Component } from "@angular/core";
import { Router, ActivatedRoute } from "@angular/router";

@Component({
  selector: "summary",
  templateUrl: "./summary.component.html"
})
export class Summary {

  constructor(private activatedRoute: ActivatedRoute) {}

  get id(): string {
    return this.activatedRoute.snapshot.params["id"];
  }
}
```

Компонент `Summary` объявляет зависимость от объекта `ActivatedRoute`, которую Angular разрешит с помощью функции внедрения зависимостей. Класс `ActivatedRoute` отвечает за описание текущего маршрута через свойство `snapshot`. Компонент `Summary` считывает значение параметра с именем `id`, который будет содержать идентификатор заказа. Например, для URL вида `/summary/5` значение параметра `id` будет равно 5. Чтобы задать шаблон для компонента, добавьте в папку `src/app` файл с именем `summary.component.html` с содержанием, как в листинге 19.5.

Листинг 19.5. Содержимое файла `summary.component.html` из папки `src/app`

```
<div class="m-2 text-center">
  <h2>Thanks!</h2>
  <p>Thanks for placing your order.</p>
  <p>Your order is #{{ id }}</p>
  <p>We'll ship your goods as soon as possible.</p>
  <button class="btn btn-primary" routerLink="/products">OK</button>
</div>
```

Шаблон отображает значение свойства `id`, полученное из активного маршрута, и представляет элемент кнопки `button`, при нажатии на которую осуществляется переход к URL `/products`.

19.2.2. Создание конфигурации маршрутизации

Чтобы описать поддерживаемые приложением URL-адреса и компоненты, которые должны отображаться для каждого из них, внесите изменения, показанные в листинге 19.6, для создания конфигурации системы маршрутизации Angular.

Листинг 19.6. Настройка приложения в файле app.module.ts из папки src/app

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from "@angular/forms";
import { DataModelModule } from "./data/data.module";
import { ProductItem } from './productItem.component';
import { CategegoryList } from "./categoryList.component";
import { Header } from "./header.component";
import { ProductList } from "./productList.component";
import { RouterModule } from '@angular/router'
import { OrderDetails } from "./orderDetails.component";
import { Summary } from "./summary.component";

const routes = RouterModule.forRoot([
  { path: "products", component: ProductList },
  { path: "order", component: OrderDetails},
  { path: "summary/:id", component: Summary},
  { path: "", redirectTo: "/products", pathMatch: "full"}
]);

@NgModule({
  declarations: [AppComponent, ProductItem, CategegoryList,
    Header, ProductList, OrderDetails, Summary],
  imports: [BrowserModule, AppRoutingModule, FormsModule,
    DataModelModule, routes],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Метод `RouterModule.forRoot` используется для описания URL-адресов и компонентов, которые они будут отображать, а также для указания по перенаправлению URL-адреса по умолчанию на `/products`. Чтобы сообщить Angular, куда выводить компоненты, указанные в конфигурации маршрутизации, замените содержимое файла `app.component.html` элементом из листинга 19.7.

Листинг 19.7. Замена содержимого файла app.component.html в папке src/app

```
<router-outlet></router-outlet>
```

Последнее изменение заключается в преобразовании компонента `ProductList` таким образом, чтобы его метод `submit` использовал функцию маршрутизации Angular для перехода по URL `/order`, как показано в листинге 19.8.

Листинг 19.8. Навигация в файле productList.component.ts из папки src/app

```

import { Component } from "@angular/core";
import { DataSource } from './data(dataSource';
import { Product } from './data/entities';
import { Router } from '@angular/router';

@Component({
  selector: "product-list",
  templateUrl: "./productList.component.html"
})

```

```

export class ProductList {
    selectedCategory = "All";

    constructor(public dataSource: DataSource, private router: Router) {}

    get products(): Product[] {
        return this.dataSource.getProducts("id",
            this.selectedCategory === "All"
                ? undefined : this.selectedCategory);
    }

    get categories(): string[] {
        return ["All", ...this.dataSource.getCategories()];
    }

    handleCategorySelect(category: string) {
        this.selectedCategory = category;
    }

    handleAdd(data: {product: Product, quantity: number}) {
        this.dataSource.order.addProduct(data.product, data.quantity);
    }

    handleSubmit() {
        this.router.navigateByUrl("/order");
    }
}

```

Сохраните изменения и дождитесь, пока инструменты разработки перестроят приложение и перезагрузят браузер. Наше приложение завершено. Теперь вы сможете выбирать товары, просматривать сводку заказа и отправлять ее на сервер, как показано на рис. 19.2.

СОВЕТ

Если при нажатии кнопки «Отправить заказ» (Submit Order) меняется только URL в браузере, то, скорее всего, причина в том, что вы не заменили содержимое файла `app.component.html`, как показано в листинге 19.7.

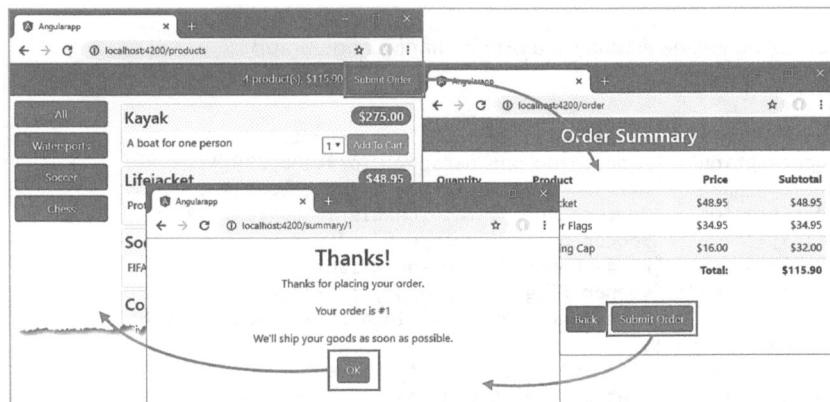


Рис. 19.2. Добавление компонентов в пример приложения

19.3. РАЗВЕРТЫВАНИЕ ПРИЛОЖЕНИЯ

Инструменты разработки Angular полагаются на сервер разработки Webpack, который не подходит для размещения производственного приложения, поскольку добавляет в генерируемые им пакеты JavaScript функцию автоматической перезагрузки. В этом разделе мы рассмотрим процесс подготовки приложения на Angular к развертыванию, который применим к любому другому веб-приложению.

19.3.1. Добавление пакета производственного HTTP-сервера

Для производственной версии приложения требуется обычный HTTP-сервер для доставки файлов HTML, CSS и JavaScript в браузер. Для нашего примера мы воспользуемся все тем же сервером Express, который используется для всех примеров в этой части книги и который является хорошим выбором для любого веб-приложения. Нажатием **Control+C** остановите работу Angular и в папке `angularapp` выполните команду из листинга 19.9 для установки пакета `express`.

Вторая команда устанавливает пакет `connect-history-api-fallback`. Он полезен при развертывании приложений, использующих маршрутизацию URL, а также сопоставляет запросы к URL, которые поддерживает приложение, с файлом `index.html`. Это гарантирует, что при перезагрузке браузера пользователь не получит ошибку `not found`.

Листинг 19.9. Добавление пакетов для развертывания

```
npm install --save-dev express@4.18.2
npm install --save-dev connect-history-api-fallback@2.0.0
```

19.3.2. Создание файла постоянного хранения данных

Чтобы создать файл постоянного хранения данных для веб-сервиса, добавьте в папку `angularapp` файл с именем `data.json` и поместите в него содержимое из листинга 19.10.

Листинг 19.10. Содержимое файла `data.json` из папки `angularapp`

```
{
  "products": [
    { "id": 1, "name": "Kayak", "category": "Watersports",
      "description": "A boat for one person", "price": 275 },
    { "id": 2, "name": "Lifejacket", "category": "Watersports",
      "description": "Protective and fashionable", "price": 48.95 },
    { "id": 3, "name": "Soccer Ball", "category": "Soccer",
      "description": "FIFA-approved size and weight", "price": 19.50 },
    { "id": 4, "name": "Corner Flags", "category": "Soccer",
      "description": "Give your playing field a professional touch",
      "price": 34.95 },
    { "id": 5, "name": "Stadium", "category": "Soccer",
      "description": "Flat-packed 35,000-seat stadium", "price": 79500 },
    { "id": 6, "name": "Thinking Cap", "category": "Chess",
      "description": "Improve brain efficiency by 75%", "price": 16 },
```

```

    { "id": 7, "name": "Unsteady Chair", "category": "Chess",
      "description": "Secretly give your opponent a disadvantage",
      "price": 29.95 },
    { "id": 8, "name": "Human Chess Board", "category": "Chess",
      "description": "A fun game for the family", "price": 75 },
    { "id": 9, "name": "Bling Bling King", "category": "Chess",
      "description": "Gold-plated, diamond-studded King", "price": 1200 }
  ],
  "orders": []
}

```

19.3.3. Создание сервера

Для создания сервера, который будет доставлять приложение и его данные в браузер, создайте в папке angularapp файл `server.js` и поместите в него код, показанный в листинге 19.11.

Листинг 19.11. Содержимое файла server.js из папки angularapp

```

const express = require("express");
const jsonServer = require("json-server");
const history = require("connect-history-api-fallback");

const app = express();
app.use(history());
app.use("/", express.static("dist/angularapp"));

const router = jsonServer.router("data.json");
app.use(jsonServer.bodyParser)
app.use("/api", (req, resp, next) => router(req, resp, next));

const port = process.argv[3] || 4001;
app.listen(port, () => console.log('Running on port ${port}'));

```

Операторы в файле `server.js` настраивают пакеты `express` и `json-server` на обслуживание содержимого каталога `dist/angularapp`, куда в процессе сборки Angular поместит JavaScript-пакеты приложения и HTML-файл, указывающий браузеру на необходимость их загрузки. Все URL с префиксом `/api` будут обрабатываться веб-сервисом.

19.3.4. Использование относительных URL для запросов данных

Веб-сервис, предоставляющий приложению данные, работает параллельно с сервером разработки Angular. Чтобы подготовиться к отправке запросов на один порт, измените класс `RemoteDataSource` в соответствии с листингом 19.12.

Листинг 19.12. Относительные URL-адреса в файле `remoteDataSource.ts` из папки `src/app/data`

```

import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";

```

```

import { map } from "rxjs/operators";
import { DataSourceImpl } from "./dataSource";
import { Product, Order } from "./entities";

// const protocol = "http";
// const hostname = "localhost";
// const port = 4600;

const urls = {
    // products: `${protocol}://${hostname}:${port}/products`,
    // orders: `${protocol}://${hostname}:${port}/orders`
    products: "/api/products",
    orders: "/api/orders"
};

@Injectable()
export class RemoteDataSource extends DataSourceImpl {

    constructor(private http: HttpClient) {
        super();
    }

    loadProducts(): Observable<Product[]> {
        return this.http.get<Product[]>(urls.products);
    }

    storeOrder(order: Order): Observable<number> {
        let orderData = {
            lines: [...order.orderLines.values()].map(ol => ({
                productId: ol.product.id,
                productName: ol.product.name,
                quantity: ol.quantity
            }))
        }
        return this.http.post<{ id: number }>(urls.orders, orderData)
            .pipe<number>(map(val => val.id));
    }
}

```

URL-адреса в листинге 19.12 указаны относительно URL-адреса, используемого для запроса HTML-документа, следуя общепринятой конвенции, согласно которой запросы данных начинаются с префикса /api.

19.3.5. Сборка приложения

Для создания производственной сборки приложения перед развертыванием выполните команду из листинга 19.13 в папке angularapp.

Листинг 19.13. Создание производственной сборки

```
ng build --configuration "production"
```

В процессе сборки в папке dist создается набор оптимизированных файлов. Данный процесс может занять несколько минут. В результате вы увидите следующее сообщение, где показано, какие файлы были включены в пакет:

```

Browser application bundle generation complete.
Copying assets complete.
Generating index.html... 1 rules skipped due to selector errors:
  legend+* -> Cannot read properties of undefined (reading 'type')
Index.html generation complete.

Initial Chunk Files      | Names      | Raw Size | Size
main.16225861184cae00.js | main       | 246.35 kB | 64.12 kB
styles.9c36b9530393e161.css | styles     | 187.50 kB | 19.44 kB
polyfills.89ae3309894ba767.js | polyfills  | 33.09 kB | 1.70 kB
runtime.2d99e508040b4ce1.js | runtime    | 898 bytes | 518 bytes

| Total      | 467.82 kB | 94.77 kB

Build at: 2023-03-26T18:20:11.031Z - Hash: d8caebf2a4448e24 - Time: 20106ms

```

19.3.6. Тестирование производственной сборки

Чтобы убедиться, что процесс сборки прошел успешно и изменения конфигурации вступили в силу, выполните в папке angularapp команду из листинга 19.14.

Листинг 19.14. Запуск сервера

```
node server.js
```

После чего вы увидите следующий результат:

```
Running on port 4001
```

Откройте браузер и перейдите по адресу <http://localhost:4001>, где будет показано приложение, как на рис. 19.3.

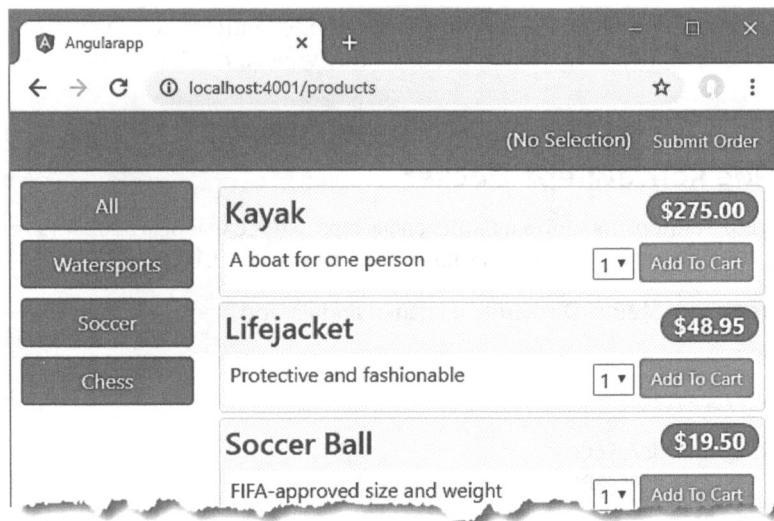


Рис. 19.3. Запуск производственной сборки

19.4. КОНТЕЙНЕРИЗАЦИЯ ПРИЛОЖЕНИЯ

В завершение этой главы мы создадим контейнер Docker для нашего Angular-приложения, чтобы его можно было развернуть в производственной среде. Если вы не установили Docker в главе 17, то вам необходимо сделать это сейчас, чтобы выполнить остальные примеры.

19.4.1. Подготовка приложения

Первым делом мы сформируем конфигурационный файл для NPM, который будет использоваться для загрузки дополнительных пакетов, необходимых приложению для работы в контейнере. Создайте в папке angularapp файл `deploy-package.json` с содержимым, показанным в листинге 19.15.

Листинг 19.15. Содержимое файла `deploy-package.json` из папки angularapp

```
{
  "name": "angularapp",
  "description": "Angular Web App",
  "repository": "https://github.com/manningbooks/essential-typescript-5",
  "license": "BSD",
  "devDependencies": {
    "express": "4.18.2",
    "json-server": "0.17.3",
    "connect-history-api-fallback": "2.0.0"
  }
}
```

В разделе `devDependencies` указываются пакеты, необходимые для запуска приложения в контейнере. Все пакеты, для которых в кодовых файлах приложения имеются операторы `import`, будут включены в пакет, созданный `webpack`, и перечислены в виде списка. Остальные поля описывают приложение и используются преимущественно для предотвращения появления предупреждений при создании контейнера.

19.4.2. Создание контейнера Docker

Чтобы определить контейнер, добавьте в папку angularapp файл с именем `Dockerfile` (без расширения), содержимое которого показано в листинге 19.16.

Листинг 19.16. Содержимое файла `Dockerfile` из папки angularapp

```
FROM node:18.14.0

RUN mkdir -p /usr/src/angularapp

COPY dist /usr/src/angularapp/dist/
COPY data.json /usr/src/angularapp/
COPY server.js /usr/src/angularapp/
COPY deploy-package.json /usr/src/angularapp/package.json

WORKDIR /usr/src/angularapp
```

```
RUN echo 'package-lock=false' >> .npmrc
RUN npm install
EXPOSE 4001
CMD [ "node", "server.js" ]
```

Содержимое `Dockerfile` использует базовый образ, сконфигурированный с помощью `Node.js`, который копирует файлы, необходимые для запуска приложения, в контейнер, а также файл со списком пакетов, требуемых для развертывания.

Чтобы ускорить процесс контейнеризации, создайте в папке `angularapp` файл `.dockerignore` и поместите в него утверждение из листинга 19.17. Теперь Docker будет игнорировать папку `node_modules`, которая не нужна в контейнере и отнимает много времени при обработке.

Листинг 19.17. Содержимое файла .dockerignore из папки angularapp

```
node_modules
```

Выполните команду из листинга 19.18 в папке `angularapp`, чтобы создать образ приложения вместе со всеми необходимыми пакетами.

Листинг 19.18. Сборка образа Docker

```
docker build . -t angularapp -f Dockerfile
```

Образ – это шаблон для контейнеров. По мере того как Docker обрабатывает инструкции в файле Docker, загружаются и устанавливаются пакеты NPM, а файлы конфигурации и кода скопируются в образ.

19.4.3. Запуск приложения

После формирования образа создайте и запустите новый контейнер с помощью команды, показанной в листинге 19.19.

Листинг 19.19. Запуск контейнера Docker

```
docker run -p 4001:4001 angularapp
```

Вы можете протестировать приложение, открыв страницу `http://localhost:4000` в браузере, где будет отображен ответ, выданный веб-сервером, запущенным в контейнере (рис. 19.4).

Для вывода списка запущенных контейнеров Docker на текущей машине выполните команду из листинга 19.20.

Листинг 19.20. Вывод списка контейнеров

```
docker ps
```

Вы увидите примерно следующий список (для краткости я опустил некоторые поля):

CONTAINER ID	IMAGE	COMMAND	CREATED
48dbd2431700	angularapp	"docker-entry"	41 seconds ago

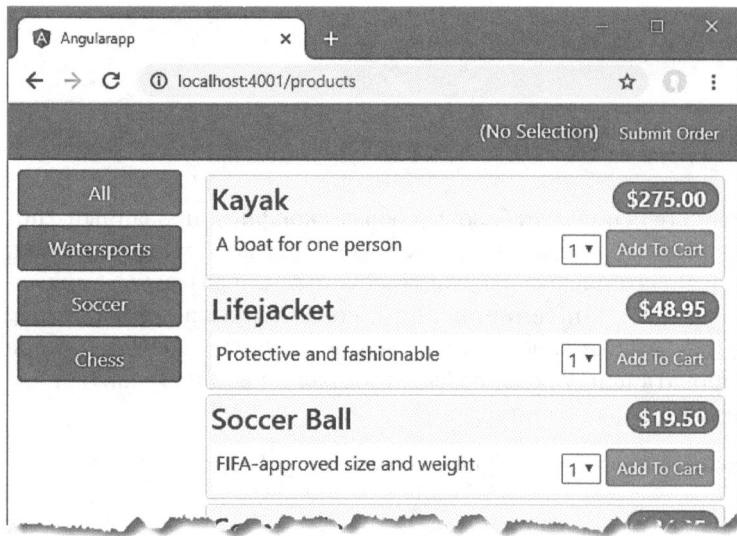


Рис. 19.4. Запуск приложения в контейнере

Используя значение в столбце `CONTAINER ID`, выполните команду, показанную в листинге 19.21, чтобы остановить работу контейнера.

Листинг 19.21. Остановка контейнера

```
docker stop 48dbd2431700
```

Теперь наше Angular-приложение готово к развертыванию на любой платформе, поддерживающей Docker.

РЕЗЮМЕ

В этой главе мы завершили работу над примером приложения Angular, добавив новые компоненты и использовав функционал маршрутизации URL для определения того, когда они будут отображаться пользователю. Мы подготовили производственную сборку приложения и создали контейнер для его удобного развертывания.

- Angular предоставляет инструменты разработки, необходимые для создания производственной версии приложения и подготовки его к развертыванию.
- Как и любое другое приложение TypeScript, проекты на Angular компилируются в чистый JavaScript и могут быть развернуты с помощью стандартных инструментов и контейнеров.

В следующей главе мы напишем веб-приложение с использованием фреймворка React.

Создание приложения на React, часть 1

В этой главе

- ✓ Создание проекта на React.
- ✓ Настройка компонентов React с использованием формата JSX.
- ✓ Создание компонентов на основе классов и функций.
- ✓ Создание хранилища данных, в котором хранятся локальные данные и используется HTTP API.

В текущей главе мы начнем создавать React-приложение, которое будет иметь тот же функционал, что и примеры из предыдущих глав. Язык TypeScript является необязательным при разработке на React, однако из-за хорошей его поддержки обеспечивается хороший рабочий процесс. В качестве краткой справки в табл. 20.1 перечислены параметры компилятора TypeScript, используемые в этой главе.

Таблица 20.1. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
<code>allowJs</code>	Включает в процесс компиляции файлы JavaScript
<code>allowSyntheticDefaultImports</code>	Разрешает импорт из модулей, в которых не объявлен экспорт по умолчанию. Данная опция используется для повышения совместимости кода

Продолжение ↓

Таблица 20.1 (продолжение)

Название	Описание
esModuleInterop	Добавляет вспомогательный код для импорта из модулей без экспорта по умолчанию. Используется совместно с опцией <code>allowSyntheticDefaultImports</code>
forceConsistentCasingInFileNames	Гарантирует согласованность регистра имен в операторах <code>import</code>
include	Указывает на файлы и папки, которые необходимо включить в процесс компиляции
isolatedModules	Рассматривает каждый файл как отдельный модуль (повышает совместимость с Babel)
jsx	Определяет, как обрабатываются HTML-элементы в файлах JSX/TSX
lib	Выбирает файлы деклараций типов, которые использует компилятор
module	Задает формат используемых модулей
moduleResolution	Задает стиль разрешения модуля, который следует использовать для разрешения зависимостей
noEmit	Запрещает компилятору генерировать JavaScript-код, в результате чего он проверяет код только на наличие ошибок
noFallthroughCasesInSwitch	Предотвращает неявный переход в следующий блок <code>case</code> в <code>switch</code> -конструкциях, если не указан оператор <code>break</code>
resolveJsonModule	Позволяет импортировать JSON-файлы как модули
skipLibCheck	Ускоряет компиляцию, пропуская обычную проверку файлов деклараций
strict	Включает более строгую проверку кода TypeScript
target	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

20.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

Проекты на React проще всего создавать с помощью пакета `create-react-app`. Откройте новое окно командной строки, перейдите в удобную директорию и выполните команду из листинга 20.1 для установки пакета `create-react-app`.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 20.1. Установка пакета создания проекта

```
npm install --global create-react-app@5.0.1
```

После установки пакета запустите команду, показанную в листинге 20.2, для создания проекта с именем `reactapp`.

Листинг 20.2. Создание проекта React

```
npx create-react-app reactapp --template typescript --use-npm
```

Аргумент `--template typescript` указывает пакету `create-react-app` создать проект React, настроенный для использования TypeScript, что включает в себя установку и настройку компилятора TypeScript, а также файлов декларации, описывающих React API и связанные с ним инструменты. Команда `--use-npm` устанавливает пакеты с помощью менеджера пакетов NPM.

20.1.1. Настройка веб-сервиса

После завершения процесса создания выполните команды, показанные в листинге 20.3, чтобы перейти в папку проекта, добавить пакеты, предоставляющие веб-сервис, и разрешить запуск нескольких пакетов одной командой.

Листинг 20.3. Добавление пакетов в проект

```
cd reactapp
npm install --save-dev json-server@0.17.3
npm install --save-dev npm-run-all@4.1.5
```

Чтобы предоставить данные для веб-сервиса, добавьте в папку `reactapp` файл `data.js` с утверждениями из листинга 20.4.

Листинг 20.4. Содержимое файла `data.js` из папки `reactapp`

```
module.exports = function () {
  return {
    products: [
      { id: 1, name: "Kayak", category: "Watersports",
        description: "A boat for one person", price: 275 },
      { id: 2, name: "Lifejacket", category: "Watersports",
        description: "Protective and fashionable", price: 48.95 },
      { id: 3, name: "Soccer Ball", category: "Soccer",
        description: "FIFA-approved size and weight",
        price: 19.50 },
      { id: 4, name: "Corner Flags", category: "Soccer",
        description:
          "Give your playing field a professional touch",
        price: 34.95 },
      { id: 5, name: "Stadium", category: "Soccer",
        description: "Flat-packed 35,000-seat stadium",
        price: 79500 },
      { id: 6, name: "Thinking Cap", category: "Chess",
        description: "Improve brain efficiency by 75%",
        price: 16 },
    ]
  }
}
```

```

        {
          id: 7, name: "Unsteady Chair", category: "Chess",
          description: "Secretly give your opponent a disadvantage",
          price: 29.95 },
        {
          id: 8, name: "Human Chess Board", category: "Chess",
          description: "A fun game for the family", price: 75 },
        {
          id: 9, name: "Bling Bling King", category: "Chess",
          description: "Gold-plated, diamond-studded King",
          price: 1200 }
      ],
      orders: []
    }
}

```

Обновите раздел `scripts` в файле `package.json` для настройки инструментов разработки таким образом, чтобы цепочка инструментов React и веб-сервис запускались одновременно, как показано в листинге 20.5.

Листинг 20.5. Настройка инструментов в файле package.json из папки reactapp

```

...
"scripts": {
  "json": "json-server data.js -p 4600",
  "serve": "react-scripts start",
  "start": "npm-run-all -p serve json",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
...

```

20.1.2. Установка пакета Bootstrap CSS

С помощью консоли выполните команду, показанную в листинге 20.6, в папке `reactapp` для добавления в проект CSS-фреймворка Bootstrap.

Листинг 20.6. Добавление пакета CSS

```
npm install bootstrap@5.2.3
```

Чтобы убедиться, что таблица стилей Bootstrap включена в приложение, добавьте в файл `index.tsx` в папке `src` оператор `import` из листинга 20.7.

Листинг 20.7. Объявление зависимости в файле index.tsx из папки src

```

import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import 'bootstrap/dist/css/bootstrap.css';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>

```

```
<App />
</React.StrictMode>
);
reportWebVitals();
```

20.1.3. Запуск примера приложения

С помощью командной строки выполните в папке `reactapp` команду, показанную в листинге 20.8.

Листинг 20.8. Запуск инструментов разработки

```
npm start
```

Запустятся веб-сервис и инструменты сборки React, и вы увидите следующий результат:

```
Compiled successfully!
You can now view reactapp in the browser.
  Local:          http://localhost:3000
  On Your Network:  http://172.22.208.1:3000
Note that the development build is not optimized.
To create a production build, use npm run build.
```

Откроется новое окно браузера по адресу `http://localhost:3000`, где будет показано содержимое-заполнитель, предоставленное в процессе создания проекта, как на рис. 20.1.

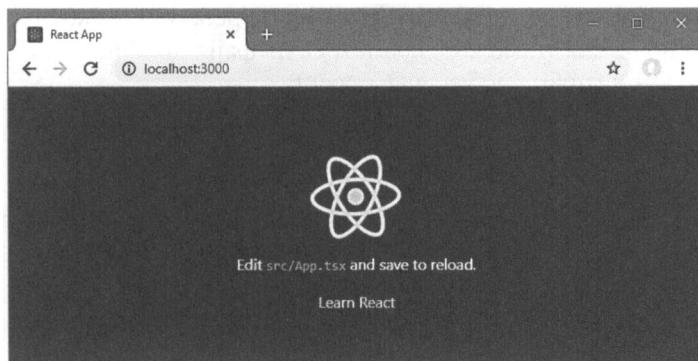


Рис. 20.1. Запуск примера приложения

20.2. РОЛЬ TYPESCRIPT В РАЗРАБОТКЕ НА REACT

В целом TypeScript необязателен при работе с React, и это отражается на способе настройки инструментов разработки и компилятора TypeScript. Для создания пакета JavaScript и доставки его в браузер используются webpack и Webpack Development.

Разработка на React базируется на формате JSX, который позволяет смешивать JavaScript и HTML в одном файле. В инструментах разработки React уже

реализована возможность преобразования JSX-файлов в чистый JavaScript с использованием пакета Babel. Babel — это компилятор JavaScript, позволяющий транслировать код, написанный на последних версиях JavaScript, в код, работающий в старых браузерах, подобно тому как это делает компилятор TypeScript. Babel расширяется с помощью подключаемых модулей — плагинов, — а его поддержка выросла до того, что теперь он может транслировать множество других форматов в JavaScript, включая JSX-файлы. На рис. 20.2 показаны основные элементы цепочки инструментов React для обычного JavaScript-проекта.

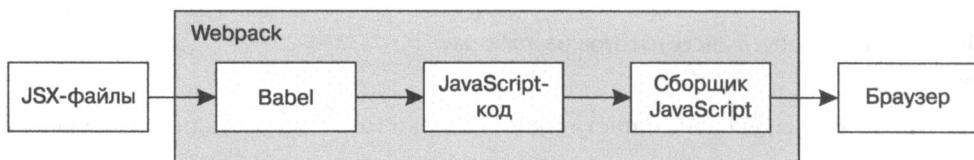


Рис. 20.2. Инструментарий разработки JavaScript React

Плагин Babel, отвечающий за JSX, выполняет ту же функцию, что и фабричный класс JSX, созданный в главе 16, заменяя HTML-фрагменты JavaScript-выражениями, используя при этом более сложный и эффективный API React. В результате преобразования получается чистый JavaScript, который упаковывается в файл, чтобы браузер смог его получить и выполнить. В этот пакет также входит код JavaScript для распаковки всех необходимых приложению ресурсов CSS или изображений.

Способ, которым инструментарий React обрабатывает TypeScript, несколько необычен, и понять, что происходит, можно, посмотрев на файл конфигурации компилятора TypeScript, который был добавлен в проект:

```
{
  "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"
]
}
```

Стоит обратить внимание на настройку `noEmit`. Когда параметр `noEmit` имеет значение `true`, компилятор TypeScript не будет генерировать JavaScript-файлы. Причина этой необычной настройки компилятора заключается в том, что за преобразование кода TypeScript в JavaScript отвечает именно пакет Babel, а не компилятор TypeScript. В состав инструментария React входит плагин Babel, который и трансформирует TypeScript в чистый JavaScript.

Babel способен преобразовывать TypeScript в JavaScript, но он не понимает особенностей TypeScript и не умеет выполнять проверку типов. Эта задача возложена на компилятор TypeScript, так что ответственность за работу с TypeScript разделена: компилятор TypeScript отвечает за выявление ошибок типа, а Babel – за создание JavaScript-кода, который будет выполняться браузером, как показано на рис. 20.3.

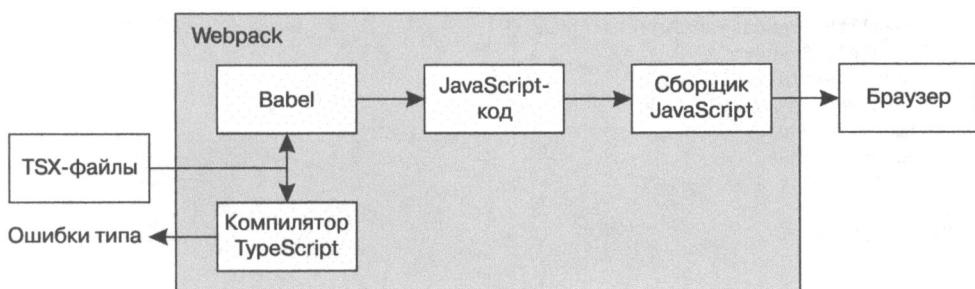


Рис. 20.3. Инструментарий разработки TypeScript React

Настройка `noEmit` в нашем случае имеет смысл, поскольку компилятору TypeScript не нужно создавать JavaScript-файлы для выполнения проверки типов.

Однако следует помнить, что подход с Babel имеет свои ограничения при работе с TypeScript, хотя их на удивление немного. На момент написания книги отсутствовала полноценная поддержка перечислений, и нельзя было использовать пространства имен. Эта возможность считается устаревшим предшественником модулей JavaScript, и рассматривать мы ее не будем.

ПРИМЕЧАНИЕ

Возможно, при запуске инструментов разработки вы могли получить предупреждение о несоответствии версий TypeScript. Это предупреждение отражает возможную разницу между функциями проверки типов, реализованными в последней версии компилятора TypeScript, и тем, как код TypeScript переводится в JavaScript с помощью Babel. Для такого простого проекта, как наш, серьезные проблемы вряд ли возникнут, однако рекомендуется использовать только те версии TypeScript, которые явно поддерживаются пакетом `create-react-app`.

Вы можете изменить конфигурацию компилятора TypeScript в соответствии с необходимыми языковыми возможностями. Например, в листинге 20.9 выбрана версия JavaScript ES2022, которая включает поддержку более современных возможностей языка, в том числе часто используемого мной оператора расширения spread.

Листинг 20.9. Изменение конфигурации в файле tsconfig.json из папки reactapp

```
{
  "compilerOptions": {
    "target": "ES2022",
    "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"
  ]
}
```

Преобразование Babel может работать с оператором spread без изменения конфигурации, а эффект от настройки `target` в листинге 20.9 лишь предотвращает появление ошибок в компиляторе TypeScript.

20.3. ОПРЕДЕЛЕНИЕ ТИПОВ СУЩНОСТЕЙ

React фокусируется на представлении HTML-контента пользователю, оставляя другие задачи, такие как управление данными приложения и выполнение HTTP-запросов, другим пакетам. Позже мы добавим пакеты в проект, чтобы дополнить набор функций, необходимых нашему примеру приложения, но пока сосредоточимся на тех функциях, которые предоставляет React. Для начала потребуется определить сущности, которые будет использовать приложение. Создайте папку `src/data` и поместите в нее файл с именем `entities.ts` с кодом из листинга 20.10.

Листинг 20.10. Содержимое файла entities.ts из папки src/data

```
export type Product = {
  id: number,
  name: string,
```

```
description: string,
category: string,
price: number
};

export type ProductSelection = {
    product: Product, quantity: number;
}

export class ProductSelectionHelpers {

    public static total(selections : ProductSelection[]) {
        return selections.reduce((total, line) =>
            total + (line.product.price * line.quantity), 0);
    }

    public static productCount(selections: ProductSelection[]) {
        return selections.reduce((total, line) => total + line.quantity, 0)
    }
}

export class ProductSelectionMutations {

    public static addProduct(selections : ProductSelection[],
        product: Product, quantity: number) {

        const index = selections
            .findIndex(line => line.product.id === product.id);
        if (index > -1) {
            selections[index].quantity += quantity;
        } else {
            selections.push({ product, quantity})
        }
    }

    public static remove(selections: ProductSelection[], id: number) {
        selections.forEach((line, index) => {
            if (line.product.id === id) {
                selections = selections.splice(index, 1);
            }
        });
    }
}
```

React и его основные пакеты поддержки лучше всего работают с простыми структурами данных, которые определяются отдельно от функций, изменяющих их. В листинге 20.10 определены типы `Product` и `ProductSelection`, а также класс `ProductSelectionHelpers`, который задает статические методы для выполнения общих вычислений над этими типами, и класс `ProductSelectionMutations`, определяющий статические методы для их изменения. Данные классы не требуются в React, который накладывает мало ограничений на структуру кода, но я считаю, что это полезный способ последовательного определения общих операций.

20.4. ОТОБРАЖЕНИЕ ОТФИЛЬТРОВАННОГО СПИСКА ТОВАРОВ

В React используется формат JSX, позволяющий задавать HTML-элементы наряду с кодом JavaScript, аналогично подходу, к которому мы прибегали при создании автономного веб-приложения. В процессе компиляции HTML-элементы преобразуются в JavaScript-инструкции, которые используют React API для эффективного отображения контента пользователю — более элегантный подход, чем тот, что был представлен в главе 16.

Ключевым строительным блоком в приложении React является компонент, отвечающий за генерацию HTML-контента. Компоненты настраиваются с помощью реквизитов. Они могут реагировать на взаимодействие с пользователем, обрабатывая события, вызываемые элементами HTML, которые они отображают, и определять локальные данные состояния.

Чтобы отобразить информацию об одном товаре, добавьте в папку `src` файл с именем `productItem.tsx` и поместите в него код, показанный в листинге 20.11, для создания простого компонента `React`.

Листинг 20.11. Содержимое файла `productItem.tsx` из папки `src`

```
import React, { Component, ChangeEvent } from "react";
import { Product } from "./data/entities";

interface Props {
    product: Product,
    callback: (product: Product, quantity: number) => void
}

interface State {
    quantity: number
}

export class ProductItem extends Component<Props, State> {

    constructor(props: Props) {
        super(props);
        this.state = {
            quantity: 1
        }
    }

    render() {
        return <div className="card m-1 p-1 bg-light">
            <h4>
                { this.props.product.name }
                <span className="badge rounded-pill bg-primary float-end">
                    ${ this.props.product.price.toFixed(2) }
                </span>
            </h4>
            <div className="card-text bg-white p-1">
                { this.props.product.description }
                <button className="btn btn-success btn-sm float-end" onClick={ this.handleAddToCart } >

```

```

        Add To Cart
    </button>
    <select className="form-control-inline float-end m-1"
            onChange={ this.handleQuantityChange }>
        <option>1</option>
        <option>2</option>
        <option>3</option>
    </select>
</div>
</div>
}

handleQuantityChange = (ev: ChangeEvent<HTMLSelectElement>): void =>
    this.setState({ quantity: Number(ev.target.value) });

handleAddToCart = (): void =>
    this.props.callback(this.props.product, this.state.quantity);
}

```

Использование TypeScript требует, чтобы типы данных, описывающие реквизиты и данные состояния, были определены и использовались в качестве аргументов обобщенного типа для класса Component. Компонент ProductItem получает реквизиты, которые предоставляет ему объект Product, и функцию обратного вызова, которая вызывается, когда пользователь нажимает кнопку Add To Cart (Добавить в корзину). У компонента ProductItem есть одно свойство данных состояния с именем quantity, которое используется для ответа, когда пользователь выбирает значение с помощью элемента select. Реквизиты и данные состояния описываются интерфейсами Props и State соответственно. Они используются в качестве параметров обобщенного типа для настройки базового класса компонентов:

```

...
export class ProductItem extends Component<Props, State> {
...

```

Аргументы обобщенного типа позволяют компилятору TypeScript проверить компонент в момент его применения, чтобы использовались только свойства, определенные интерфейсом Props, и гарантировать, что обновления будут применяться только к свойствам, определенным интерфейсом State.

Файлы деклараций для React включают типы для событий, которые HTML-элементы будут генерировать с помощью метода render. Для события change, вызванного элементом select, функция-обработчик будет получать объект ChangeEvent<HTMLSelectElement>. Изменения свойств компонента должны выполняться через метод setState, благодаря которому React узнает, что произошло обновление.

```

...
handleQuantityChange = (ev: ChangeEvent<HTMLSelectElement>): void =>
    this.setState({ quantity: Number(ev.target.value) });
...

```

Компилятор TypeScript обеспечит обработку правильного типа события и проверит, что обновления через метод setState имеют правильный тип и обновляют только те свойства, которые определены типом State.

20.4.1. Использование функциональных компонентов и хуков

Компонент в листинге 20.11 определен с использованием класса, но React также поддерживает компоненты, определенные с помощью функций, что стало наиболее популярным способом написания функций в React.

СОВЕТ

Выбор между компонентами функций и классов – вопрос личных предпочтений, и оба подхода полностью поддерживаются React. Наиболее распространенным вариантом стали хуки (hooks), но они могут вызывать затруднения в работе. Оба способа имеют свои достоинства и свободно сочетаются в проекте.

При использовании TypeScript функциональные компоненты аннотируются с помощью типа `FunctionComponent<T>`, где обобщенный тип `T` описывает реквизиты, которые будет получать компонент. В листинге 20.12 компонент `ProductItem` переопределен таким образом, чтобы он был выражен в виде функции, а не класса.

Листинг 20.12. Функциональный компонент в файле productItem.tsx из папки src

```
import React, { FunctionComponent, useState } from "react";
import { Product } from "./data/entities";

interface Props {
    product: Product,
    callback: (product: Product, quantity: number) => void
}

// interface State {
//     quantity: number
// }

export const ProductItem: FunctionComponent<Props> = (props) => {

    const [quantity, setQuantity] = useState<number>(1);

    return <div className="card m-1 p-1 bg-light">
        <h4>
            { props.product.name }

            <span className="badge rounded-pill bg-primary float-end">
                ${ props.product.price.toFixed(2) }
            </span>
        </h4>
        <div className="card-text bg-white p-1">
            { props.product.description }
            <button className="btn btn-success btn-sm float-end"
                onClick={ () => props.callback(props.product, quantity) }>
                Add To Cart
            </button>
            <select className="form-control-inline float-end m-1">

```

```
        onChange={ (ev) => setQuantity(Number(ev.target.value)) }>
          <option>1</option>
          <option>2</option>
          <option>3</option>
        </select>
      </div>
    </div>
}
```

Результатом работы функции компонента является HTML, который должен быть отображен пользователю и который определяется с помощью той же комбинации элементов и выражений, что и компоненты, основанные на классах, в их методе `render`.

Компоненты, основанные на классах, опираются на свойства и методы, доступ к которым осуществляется через `this`, чтобы реализовать данные о состоянии и участвовать в жизненном цикле, который React предоставляет приложениям. Функциональные компоненты используют функцию, называемую «хуком» (от англ. `hook`), для достижения того же результата:

```
...
const [quantity, setQuantity] = useState<number>(1);
...
```

Это пример использования хука состояния, предоставляющего функциональному компоненту свойство данных состояния, при изменении которого запускается обновление содержимого. Функция `useState` имеет аргумент обобщенного типа и начальное значение. Она возвращает свойство, которое можно прочитать, чтобы получить текущее значение, и функцию, которую можно вызвать для его изменения. В данном случае свойству присвоено имя `quantity`, а функции обновления — `setQuantity` в соответствии с общепринятым соглашением об именовании. В результате `quantity` можно использовать в выражениях для получения значения данных состояния:

```
...
onClick={ () => props.callback(props.product, quantity) }
...
```

Свойство `quantity` является константой, то есть его нельзя изменить. Вместо этого изменения должны осуществляться с помощью функции `setQuantity`, например, так:

```
...
<select className="form-control-inline float-end m-1"
  onChange={ (ev) => setQuantity(Number(ev.target.value)) }>
...
```

Использование свойств и функций по отдельности гарантирует, что все изменения данных состояния запускают процесс обновления React, а компилятор TypeScript проверяет значения, передаваемые в функцию, чтобы убедиться, что они соответствуют аргументу обобщенного типа, указанному в функции `useState`.

20.4.2. Отображение списка категорий и заголовка

Чтобы определить компонент, который будет отображать список категорий, добавьте в папку `src` файл с именем `categoryList.tsx` и содержимым, показанным в листинге 20.13.

Листинг 20.13. Содержимое файла `categoryList.tsx` из папки `src`

```
import React, { FunctionComponent } from "react";

interface Props {
    selected: string,
    categories: string[],
    selectCategory: (category: string) => void;
}

export const CategoryList: FunctionComponent<Props> = (props) => {
    return <div className="d-grid gap-2">
        { ["All", ...props.categories].map(c => {
            let btnClass = props.selected === c
                ? "btn-primary": "btn-secondary";
            return <button key={ c }
                className={ `btn btn-block ${btnClass}` }
                onClick={ () => props.selectCategory(c) }>
                { c }
            </button>
        )) }
    </div>
}
```

Для создания компонента заголовка добавьте в папку `src` файл с именем `header.tsx` и поместите в него код из листинга 20.14.

Листинг 20.14. Содержимое файла `header.tsx` из папки `src`

```
import React, { FunctionComponent } from "react";
import { ProductSelection, ProductSelectionHelpers } from "./data/entities";

interface Props {
    selections: ProductSelection[]
}

export const Header : FunctionComponent<Props> = (props) => {
    const count = ProductSelectionHelpers.productCount(props.selections);
    const total = ProductSelectionHelpers.total(props.selections);
    return <div className="p-1 bg-secondary text-white text-end">
        { count === 0 ? "(No Selection)" :
            `${ count } product(s), ' +
            '$${ total.toFixed(2)}' }
        <button className="btn btn-sm btn-primary m-1">
            Submit Order
        </button>
    </div>
}
```

20.4.3. Создание и тестирование компонентов

Для создания компонента, который будет отображать заголовок, список товаров и кнопки категорий, добавьте в папку src файл productList.tsx с кодом, показанным в листинге 20.15.

Листинг 20.15. Содержимое файла productList.tsx из папки src

```
import React, { FunctionComponent, useState } from "react";
import { Header } from "./header";
import { ProductItem } from "./productItem";
import { CategoryList } from "./categoryList";
import { Product, ProductSelection } from "./data/entities";

interface Props {
    products: Product[],
    categories: string[],
    selections: ProductSelection[],
    addToOrder: (product: Product, quantity: number) => void
}

export const ProductList: FunctionComponent<Props> = (props) => {
    const [selectedCategory, setSelectedCategory] = useState("All");

    const products = props.products.filter(p => selectedCategory === "All"
        || p.category === selectedCategory);

    return <div>
        <Header selections={ props.selections } />
        <div className="container-fluid">
            <div className="row">
                <div className="col-3 p-2">
                    <CategoryList categories={ props.categories }
                        selected={ selectedCategory }
                        selectCategory={ setSelectedCategory } />
                </div>
                <div className="col-9 p-2">
                    {
                        products.map(p =>
                            <ProductItem key={ p.id } product={ p }
                                callback={ props.addToOrder } />
                        )
                    }
                </div>
            </div>
        </div>
    </div>
}
```

Компоненты применяются с использованием пользовательских HTML-элементов, чей тег совпадает с именем класса компонента или функции. Компоненты настраиваются с помощью реквизитов, которые можно использовать для предоставления данных или функций обратного вызова (как и в главе 16, когда мы создавали собственную реализацию JSX). Компонент ProductList обеспечивает

свою функциональность путем объединения компонентов `Header`, `CategoryList` и `ProductItem`, каждый из которых конфигурируется с помощью реквизитов, получаемых компонентом `ProductList`, или данных о его состоянии.

Чтобы убедиться, что компоненты могут отображать контент пользователю, замените содержимое файла `App.tsx` на код, показанный в листинге 20.16.

Листинг 20.16. Замена содержимого файла `App.tsx` из папки `src`

```
import React, { FunctionComponent, useState } from 'react';
import { Product, ProductSelection, ProductSelectionMutations }
    from './data/entities';
import { ProductList } from './productList';

let testData: Product[] = [1, 2, 3, 4, 5].map(num =>
    ({ id: num, name: `Prod${num}`, category: `Cat${num % 2}`,
        description: `Product ${num}`, price: 100}))

export const App: FunctionComponent = () => {
    const [selections, setSelections] = useState(Array<ProductSelection>());
    const addToOrder = (product: Product, quantity: number) => {
        setSelections(curr => {
            ProductSelectionMutations.addProduct(curr, product, quantity);
            return [...curr];
        });
    };
    const categories = [...new Set(testData.map(p => p.category))];

    return <div className="App">
        <ProductList products={ testData }
            categories={categories }
            selections={ selections }
            addToOrder= { addToOrder } />
    </div>
}

export default App;
```

Компонент `App` был обновлен для отображения списка `ProductList`, который настроен с помощью тестовых данных. Хук `useState` нужен, чтобы обеспечить повторное отображение компонента только при изменении выбора товара. Позже мы добавим поддержку работы с веб-сервисом, но изменений в листинге 20.16 достаточно для отображения списка товаров, как показано на рис. 20.4. (Возможно, вам придется перезагрузить браузер, чтобы увидеть изменения, поскольку функция автоматической перезагрузки не самый надежный инструмент.)

При нажатии кнопки `Add To Cart` (Добавить в корзину) в заголовке может отобразиться больше товаров, чем ожидается. Это связано с тем, что проект работает в строгом режиме, в котором среда выполнения неоднократно выполняет действия по обнаружению проблемных изменений в компонентах. Вы можете

пренебречь эффектом от этих дополнительных операций. Они прекратятся после добавления хранилища данных в приложение в следующем разделе. Эта функция также отключается при компиляции любого проекта React, готового для реальной работы.

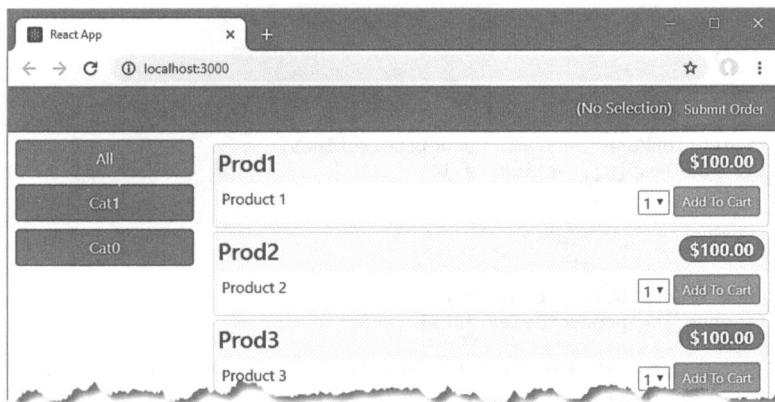


Рис. 20.4. Тестирование компонентов списка товаров

20.5. СОЗДАНИЕ ХРАНИЛИЩА ДАННЫХ

В большинстве проектов React управление данными приложения осуществляется хранилищем данных. Существует несколько пакетов для хранилища данных, но наиболее широко используется Redux. Чтобы добавить в проект пакеты Redux, откройте новое окно командной строки, перейдите в папку `reactapp` и выполните команды из листинга 20.17.

Листинг 20.17. Добавление пакетов в проект примера

```
npm install redux@4.2.1
npm install react-redux@8.0.5
npm install @reduxjs/toolkit@1.9.3
npm install --save-dev @types/react-redux@7.1.25
```

Пакет Redux включает в себя декларации TypeScript, однако для его применения необходимы дополнительные пакеты: пакет React-Redux, подключающий компоненты React к хранилищу данных, и пакет Redux Toolkit, который предоставляет стандартный способ использования Redux.

Вначале создается срез, который представляет собой комбинацию имени, некоторых исходных данных состояния и *редьюсеров* (reducer) – функций, изменяющих эти данные состояния. *Срезы* (slice) – это удобный способ создания всех функций, необходимых для управления данными в хранилище, чтобы к ним можно было обращаться в других частях приложения. Чтобы создать срез для выбранных товаров, добавьте в папку `src/data` файл `selectionSlice.ts` с кодом, показанным в листинге 20.18.

Листинг 20.18. Содержимое файла selectionSlice.ts из папки src/data

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit";
import { Product, ProductSelection, ProductSelectionMutations }
    from "./entities";

const productSelectionSlice = createSlice({
    name: "selections",
    initialState: Array<ProductSelection>(),
    reducers: {
        addToOrder(selections: ProductSelection[], action: PayloadAction<[Product, number]>) {
            ProductSelectionMutations.addProduct(selections,
                action.payload[0], action.payload[1])
        }
    }
});

export const reducer = productSelectionSlice.reducer;
export const { addToOrder } = productSelectionSlice.actions
```

Функция `createSlice` предоставляется пакетом Redux Toolkit и принимает объект со свойствами `name`, `initialState` и `reducers` (существуют и другие свойства конфигурации, но нам нужны только эти). Этот срез имеет имя `selections`, и его начальное состояние — пустой массив `ProductSelection`, и есть одна функция-редюсер с именем `addToOrder`, которая добавляет в массив выборку товара. Результат функции `createSlice` определяет свойства `reducer` и `actions`, которые должны быть экспортированы.

Следующий шаг — использование среза для создания хранилища данных. Добавьте в папку `src/data` файл с именем `dataStore.ts`, содержимое которого показано в листинге 20.19.

Листинг 20.19. Содержимое файла dataStore.ts из папки src/data

```
import { configureStore } from "@reduxjs/toolkit";

import { reducer as selectionsReducer, addToOrder }
    from "./selectionSlice";
import { TypedUseSelectorHook, useDispatch, useSelector }
    from "react-redux";

export const dataStore = configureStore({
    reducer: {
        "selections": selectionsReducer
    }
});

export type AppDispatch = typeof dataStore.dispatch;
export type RootState = ReturnType<typeof dataStore.getState>;

export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;

export const reducers = {
    addToOrder
}
```

Это базовая конфигурация хранилища данных, которая использует функции, созданные срезом, определенным в листинге 20.18, и экспортирует функции для использования в остальной части приложения.

ПРИМЕЧАНИЕ

Существует множество различных способов создания и настройки хранилища данных и его подключения к компонентам React. В текущей главе я показал самый простой подход. Важно не то, как я использую хранилище данных, а то, как я могу использовать аннотации TypeScript, чтобы описать компилятору выбранный мной подход для выполнения проверки типов.

Теперь, когда хранилище данных создано, мы можем использовать его функции для замены локального состояния, определенного компонентом App, как показано в листинге 20.20.

Листинг 20.20. Использование хранилища данных в файле App.tsx из папки src

```
import React, { FunctionComponent } from 'react';
import { Product } from './data/entities';
import { ProductList } from './productList';
import { useAppDispatch, useAppSelector, reducers }
    from "./data/dataStore";

let testData: Product[] = [1, 2, 3, 4, 5].map(num =>
    ({ id: num, name: `Prod${num}`, category: `Cat${num % 2}`,
        description: `Product ${num}`, price: 100}));

export const App: FunctionComponent = () => {

    const selections = useAppSelector(state => state.selections);
    const dispatch = useAppDispatch();

    const addToOrder = (p: Product, q: number) =>
        dispatch(reducers.addToOrder([p, q]));

    const categories = [...new Set(testData.map(p => p.category))];

    return <div className="App">
        <ProductList products={ testData }
            categories={categories }
            selections={ selections }
            addToOrder= { addToOrder } />
    </div>
}

export default App;
```

Для чтения выбранных данных из хранилища мы используем функцию useAppSelector, экспортованную в листинге 20.19, которая принимает функцию, выбирающую данные, необходимые компоненту.

Процесс внесения изменений несколько неудобен. Сначала вызывается функция useAppDispatch, возвращающая функцию, которая может быть использована

для вызова редюсера. Константе с именем `dispatch` присваивается функция, возвращенная `useAppDispatch`:

```
...
const dispatch = useAppDispatch();
...
```

Чтобы выполнить изменение, используется функция, присвоенная `dispatch`, для вызова редюсера:

```
...
const addToOrder = (p: Product, q: number) =>
  dispatch(reducers.addToOrder([p, q]));
...
```

Обратите внимание, что аргумент редюсера `addToOrder` представляет собой кортеж TypeScript. Редюсеры принимают только один аргумент, и, хотя существует возможность введения подготовительных функций, которые способны сериализовать значения данных, наиболее простой подход — упаковать необходимые значения в кортеж. Последним шагом является применение хранилища данных, как показано в листинге 20.21.

Листинг 20.21. Применение хранилища данных в файле index.tsx из папки src

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import 'bootstrap/dist/css/bootstrap.css';
import { Provider } from 'react-redux';
import { dataStore } from './data/dataStore';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <Provider store={dataStore}>
      <App />
    </Provider>
  </React.StrictMode>
);
reportWebVitals();
```

Введение хранилища данных не меняет внешний вид приложения, однако теперь за управление данными приложения отвечает хранилище.

20.5.1. Реализация клиентов HTTP API

Может показаться, что процесс настройки хранилища данных — это большой объем работы, не приносящий особой пользы, однако Redux предлагает ряд полезных возможностей после его интеграции в проект. Одна из самых полезных

функций — способность быстро и легко создавать клиенты для использования REST-ориентированных веб-сервисов в компонентах React.

Для реализации API, который генерирует данные о товарах, добавьте в папку `src/data` файл с именем `storeApis.ts` с содержимым, показанным в листинге 20.22.

Листинг 20.22. Содержимое файла `storeApis.ts` из папки `src/data`

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { Product } from './entities';

const protocol = "http";
const hostname = "localhost";
const port = 4600;

const baseUrl = `${protocol}://${hostname}:${port}`;

export const productsApi = createApi({
    reducerPath: "products",
    baseQuery: fetchBaseQuery({baseUrl}),
    endpoints: (builder) => ({
        getProducts: builder.query<Product[], void>({
            query: () => "products"
        })
    })
})

export const { useGetProductsQuery } = productsApi;
```

Функция `createApi`, предоставляемая Redux Toolkit, принимает объект конфигурации, описывающий HTTP API и определяющий функциональность, которая будет представлена остальной части приложения на React. Данная конфигурация проста: она передает приложению хук `useGetProductsQuery`, который будет отправлять HTTP-запрос GET к HTTP-сервису, созданному в начале главы, и выдавать результаты в виде массива объектов `Product`.

Листинг 20.23 обновляет хранилище данных, чтобы включить в него объекты, созданные функцией `createApi`.

Листинг 20.23. Добавление API в файл `dataStore.ts` из папки `src/data`

```
import { configureStore } from "@reduxjs/toolkit";
import { reducer as selectionsReducer, addToOrder } from "./selectionSlice";
import { TypedUseSelectorHook, useDispatch, useSelector }
    from "react-redux";
import { productsApi, useGetProductsQuery } from "./storeApis";

export const dataStore = configureStore({
    reducer: {
        "selections": selectionsReducer,
        [productsApi.reducerPath]: productsApi.reducer,
    },
    middleware: (getDefaultMiddleware) =>
        getDefaultMiddleware()
            .concat(productsApi.middleware)
});
});
```

```

export type AppDispatch = typeof dataStore.dispatch;
export type RootState = ReturnType<typeof dataStore.getState>;

export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;

export const reducers = {
    addToOrder
}

export const queries = {
    useGetProductsQuery
}

```

Эти изменения добавляют редюсеры API в хранилище данных и настраивают его на использование возможностей API, которые более подробно описаны на сайте <https://redux-toolkit.js.org/api/configureStore>. А также реэкспортируют функцию `useGetProductsQuery` из модуля API, чтобы компонентам приложения не пришлось импортировать ее напрямую из других файлов.

В листинге 20.24 тестовые данные из предыдущих примеров заменены данными, полученными через хранилище данных.

Листинг 20.24. Использование хранилища данных в файле App.tsx из папки src

```

import React, { FunctionComponent, useMemo } from 'react';
import { Product } from './data/entities';
import { ProductList } from './productList';
import { useAppDispatch, useAppSelector, reducers, queries }
    from './data/dataStore';

// let testData: Product[] = [1, 2, 3, 4, 5].map(num =>
//     ({ id: num, name: `Prod${num}`, category: `Cat${num % 2}`,
//         description: `Product ${num}`, price: 100}))

export const App: FunctionComponent = () => {

    const selections = useAppSelector(state => state.selections);
    const dispatch = useAppDispatch();

    const { data } = queries.useGetProductsQuery();

    const addToOrder = (p: Product, q: number) =>
        dispatch(reducers.addToOrder([p, q]));

    const categories = useMemo<string[]>(() => {
        return [...new Set(data?.map(p => p.category))]
    }, [data]);

    return <div className="App">
        <ProductList products={ data ?? [] }
            categories={categories }
            selections={ selections }
            addToOrder= { addToOrder } />
    </div>
}

export default App;

```

Тестовые данные заменяются результатами, полученными с помощью функции `useGetProductsQuery`. Это самый простой способ получить данные из API. Однако он не позволяет выполнять много полезных функций, таких как уведомление о загрузке и хорошая обработка ошибок.

Теперь, когда мы работаем с удаленными данными, давайте внесем некоторые корректизы в этот компонент. Во-первых, между запуском приложения и получением данных из веб-сервиса будет существовать задержка. Воспользуемся простейшим подходом, чтобы при отсутствии данных отображался пустой массив:

```
...
<ProductList products={ data ?? [] } categories={categories }
  selections={ selections } addToOrder= { addToOrder } />
...
...
```

В Redux предусмотрены функции для более эффективного управления подобным переходом, но для нашего простого примера приложения достаточно и этого. Еще одно внесенное изменение касается способа создания списка категорий:

```
...
const categories = useMemo<string[]>(() => {
  return [...new Set(data?.map(p => p.category))]

}, [data]);
...
```

Хук `useMemo` принимает функцию, которая генерирует значение, а также набор зависимостей. Функция вызывается только при изменении одной из зависимостей, что позволяет не выполнять операции при каждом отображении компонента.

В результате данные запрашиваются с сервера и добавляются в хранилище данных, что вызывает обновление, в результате которого подключенные компоненты отображают новые данные, как показано на рис. 20.5.

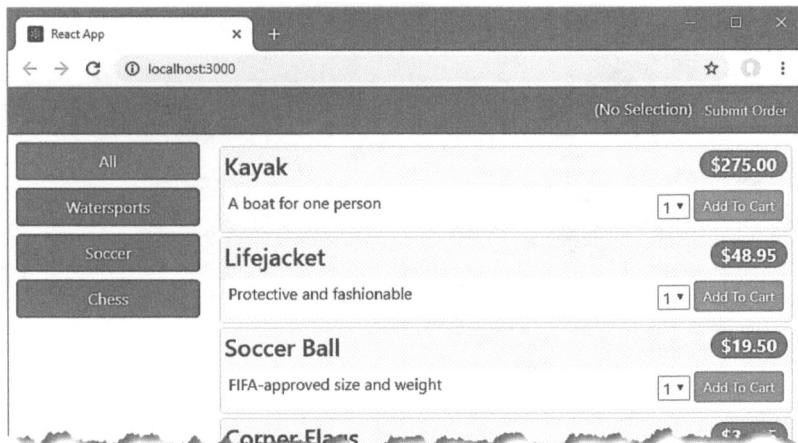


Рис. 20.5. Использование хранилища данных

РЕЗЮМЕ

В этой главе мы начали работу над проектом React, использующим TypeScript. Вы узнали о необычной конфигурации инструментов разработчика и ее влиянии на параметры компилятора TypeScript. Мы создали компоненты React, определяемые с помощью функций TypeScript, и подключили их к простому хранилищу данных Redux.

- Компоненты React используют формат JSX и отвечают за рендеринг HTML.
- Компоненты могут быть определены как классы или функции с хуками.
- Redux – наиболее популярный инструмент для хранения данных и настраивается с помощью пакета Redux Toolkit.
- Хранилища данных могут быть настроены на управление локальными данными или на использование HTTP API.

В следующей главе мы приступим к финальной части в разработке проекта на React и приготовим приложение к развертыванию.

Создание приложения на React, часть 2

В этой главе

- ✓ Использование текущего URL-адреса для выбора компонентов.
- ✓ Применение веб-сервиса заказов.
- ✓ Создание сервера развертывания и постоянного хранилища данных.
- ✓ Развёртывание приложения в контейнере.

В текущей главе мы завершим создание веб-приложения на React, добавив маршрутизацию URL и остальные компоненты, а затем подготовим проект к развертыванию в контейнере. В качестве краткой справки в табл. 21.1 перечислены параметры компилятора TypeScript, используемые в этой главе.

Таблица 21.1. Параметры компилятора TypeScript, используемые в данной главе

Название	Описание
allowJs	Включает в процесс компиляции файлы JavaScript
allowSyntheticDefaultImports	Разрешает импорт из модулей, в которых не объявлен экспорт по умолчанию. Данная опция используется для повышения совместимости кода
esModuleInterop	Добавляет вспомогательный код для импорта из модулей без экспорта по умолчанию. Используется совместно с опцией allowSyntheticDefaultImports

Продолжение ↗

Таблица 21.1 (продолжение)

Название	Описание
<code>forceConsistentCasingInFileNames</code>	Гарантирует согласованность регистра имен в операторах <code>import</code>
<code>include</code>	Указывает на файлы и папки, которые необходимо включить в процесс компиляции
<code>isolatedModules</code>	Рассматривает каждый файл как отдельный модуль (повышает совместимость с Babel)
<code>jsx</code>	Определяет, как обрабатываются HTML-элементы в файлах JSX/TSX
<code>lib</code>	Выбирает файлы деклараций типов, которые использует компилятор
<code>module</code>	Задает формат используемых модулей
<code>moduleResolution</code>	Задает стиль разрешения модуля, который следует использовать для разрешения зависимостей
<code>noEmit</code>	Запрещает компилятору генерировать JavaScript-код, в результате чего он проверяет код только на наличие ошибок
<code>noFallthroughCasesInSwitch</code>	Предотвращает неявный переход в следующий блок <code>case</code> в <code>switch</code> -конструкциях, если не указан оператор <code>break</code>
<code>resolveJsonModule</code>	Позволяет импортировать JSON-файлы как модули
<code>skipLibCheck</code>	Ускоряет компиляцию, пропуская обычную проверку файлов деклараций
<code>strict</code>	Включает более строгую проверку кода TypeScript
<code>target</code>	Задает версию языка JavaScript, которую будет использовать компилятор при генерации вывода

21.1. ПЕРВОНАЧАЛЬНЫЕ ПРИГОТОВЛЕНИЯ

В данной главе мы продолжим работать с проектом `reactapp`, начатым в главе 20. Откройте командную строку, перейдите в папку `reactapp` и выполните команду из листинга 21.1, чтобы запустить веб-сервис и инструменты разработки React.

СОВЕТ

Пример проекта для этой и остальных глав книги можно загрузить с сайта <https://github.com/manningbooks/essential-typescript-5>.

Листинг 21.1. Запуск средств разработки

```
npm start
```

После завершения процесса сборки откроется новое окно браузера, в котором будет отображен пример приложения, как показано на рис. 21.1.

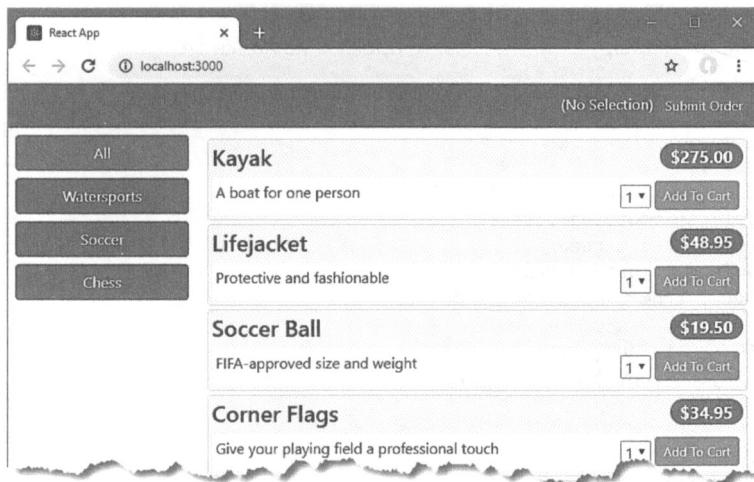


Рис. 21.1. Запуск примера приложения

21.2. НАСТРОЙКА МАРШРУТИЗАЦИИ URL-АДРЕСОВ

В большинстве реальных проектов React используется маршрутизация URL, которая использует текущий URL браузера для выбора компонентов, отображаемых пользователю. В React нет встроенной поддержки маршрутизации URL, но наиболее часто используемым пакетом является React Router. Откройте консоль, перейдите в папку `reactapp` и выполните команды, показанные в листинге 21.2, чтобы установить пакет React Router.

Листинг 21.2. Добавление пакета в проект

```
npm install react-router-dom@6.10.0
```

Пакет React Router поддерживает различные системы навигации, а пакет `react-router-dom` содержит функционал, необходимый для веб-приложений. В табл. 21.2 приведены URL-адреса, поддерживаемые нашим приложением, с кратким описанием.

Таблица 21.2. URL-адреса, поддерживаемые приложением

Название	Описание
/	Вызывает перенаправление на /products
/products	Отобразит компонент <code>ProductList</code> , определенный в главе 20
/order	Выведет компонент, отображающий детали заказа
/summary	Отобразит краткую информацию о заказе после его отправки на сервер. В URL будет включен номер заказа. Например, если ID заказа — 5, то URL будет /summary/5

Компонент `BrowserRouter` используется для организации маршрутизации приложений, основанных на браузере, и обычно добавляется в верхнюю часть иерархии компонентов, как показано в листинге 21.3.

Листинг 21.3. Добавление маршрутизации в файл index.tsx из папки src

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import 'bootstrap/dist/css/bootstrap.css';
import { Provider } from 'react-redux';
import { dataStore } from './data/dataStore';
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <Provider store={dataStore}>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </Provider>
  </React.StrictMode>
);
reportWebVitals();
```

Отдельные маршруты, используемые приложением, могут быть определены там, где это необходимо. Не все компоненты, требуемые приложению, были написаны, поэтому в листинге 21.4 задана конфигурация для URL `/products` и `/`, а остальные мы определим в последующих разделах.

Листинг 21.4. Настройка маршрутизации URL в файле App.tsx из папки src

```
import React, { FunctionComponent, useMemo } from 'react';
import { Product } from './data/entities';
import { ProductList } from './productList';
import { useAppDispatch, useAppSelector, reducers, queries }
  from "./data/dataStore";
import { Routes, Route, Navigate } from "react-router-dom";

export const App: FunctionComponent = () => {

  const selections = useAppSelector(state => state.selections);
  const dispatch = useAppDispatch();

  const { data } = queries.useGetProductsQuery();

  const addToOrder = (p: Product, q: number) =>
    dispatch(reducers.addToOrder([p, q]));
```

```

const categories = useMemo<string[]>(() => {
  return [...new Set(data?.map(p => p.category))]
}, [data]);

return (
  <div className="App">
    <Routes>
      <Route path="/products" element={
        <ProductList products={ data ?? [] }
          categories={categories}
          selections={ selections }
          addToOrder= { addToOrder } />
      }/>
      <Route path="/" element={
        <Navigate replace to="/products" />
      } />
    </Routes>
  </div>
)
}

export default App;

```

Пакет React Router использует компоненты для настройки. Компонент `Routes` содержит ряд компонентов `Route`, каждый из которых соответствует URL-пути к содержимому.

В листинге 21.4 представлены два маршрута, причем оба они описаны с помощью компонента `Route`. Первый соответствует пути `/product` и отображает компонент `ProductList`, а второй — любому пути, но использует компонент `Navigate` для перенаправления на путь `/product`. Компонент `Navigate` поставляется в составе пакета React Router. После сохранения изменений приложение будет перестроено и браузер будет перенаправлен на URL `/products`, как показано на рис. 21.2.

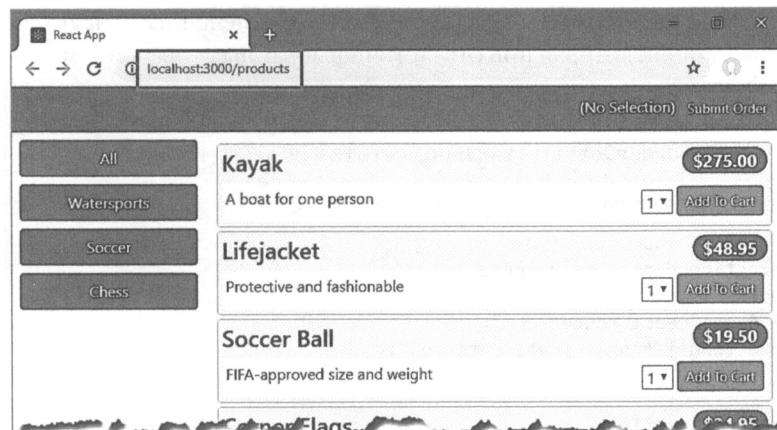


Рис. 21.2. Добавление маршрутизации URL-адресов

21.3. ЗАВЕРШЕНИЕ ФУНКЦИЙ ПРИМЕРА ПРИЛОЖЕНИЯ

Теперь, когда приложение может отображать компоненты на основе текущего URL, мы можем добавить в проект оставшиеся компоненты. Чтобы реализовать навигацию по URL с помощью кнопки, отображаемой компонентом `Header`, добавьте в файл `header.tsx` утверждения из листинга 21.5.

Листинг 21.5. Добавление навигации в файл `header.tsx` из папки `src`

```
import React, { FunctionComponent } from "react";
import { ProductSelection, ProductSelectionHelpers } from "./data/entities";
import { NavLink } from "react-router-dom";

interface Props {
    selections: ProductSelection[]
}

export const Header : FunctionComponent<Props> = (props) => {
    const count = ProductSelectionHelpers.productCount(props.selections);
    const total = ProductSelectionHelpers.total(props.selections);
    return <div className="p-1 bg-secondary text-white text-end">
        { count === 0 ? "(No Selection)"
            : `${ count } product(s), ` +
            ` ${ total.toFixed(2)}` }
        { count > 0 ?
            <NavLink to="/order" className="btn btn-sm btn-primary m-1">
                Submit Order
            </NavLink>
            : <button disabled className="btn btn-sm btn-primary m-1">
                Submit Order
            </button>
        }
    </div>
}
```

Компонент `NavLink` создает якорь (элемент с тегом `a`), который, если щелкнуть на нем, переходит на указанный URL. Классы Bootstrap, примененные к `NavLink`, придают ссылке вид кнопки, которая при отсутствии выбранных товаров заменяется на неактивный элемент `button`.

Чтобы пользователь смог увидеть детали заказа по выбранным товарам, добавьте в папку `src` файл с именем `orderDetails.tsx` и поместите в него код из листинга 21.6.

Листинг 21.6. Содержимое файла `orderDetails.tsx` из папки `src`

```
import React, { FunctionComponent } from "react";
import { ProductSelectionHelpers as Helpers }
    from "./data/entities";
import { NavLink } from "react-router-dom";
import { ProductSelection } from "./data/entities";

interface Props {
    selections: ProductSelection[],
    submitCallback: () => void
}
```

```

export const OrderDetails: FunctionComponent<Props> = (props) => {

    return <div>
        <h3 className="text-center bg-primary text-white p-2">
            Order Summary
        </h3>
        <div className="p-3">
            <table className="table table-sm table-striped">
                <thead>
                    <tr>
                        <th>Quantity</th><th>Product</th>
                        <th className="text-right">Price</th>
                        <th className="text-right">Subtotal</th>
                    </tr>
                </thead>
                <tbody>
                    { props.selections.map(selection =>
                        <tr key={ selection.product.id }>
                            <td>{ selection.quantity }</td>
                            <td>{ selection.product.name }</td>
                            <td className="text-right">
                                ${ selection.product.price.toFixed(2) }
                            </td>
                            <td className="text-right">
                                ${ Helpers.total([selection]).toFixed(2) }
                            </td>
                        </tr>
                    )}
                </tbody>
                <tfoot>
                    <tr>
                        <th className="text-right" colSpan={3}>Total:</th>
                        <th className="text-right">
                            ${ Helpers.total(props.selections).toFixed(2) }
                        </th>
                    </tr>
                </tfoot>
            </table>
        </div>
        <div className="text-center">
            <NavLink to="/products" className="btn btn-secondary m-1">
                Back
            </NavLink>
            <button className="btn btn-primary m-1"
                onClick={ props.submitCallback }>
                Submit Order
            </button>
        </div>
    </div>
}

```

Этот компонент получает реквизиты, содержащие выбранные товары, и функцию обратного вызова, которая вызывается для отправки заказа.

21.3.1. Добавление компонента подтверждения

Добавьте в папку src файл с именем `summary.tsx` и кодом, показанным в листинге 21.7, для вывода пользователю сообщения о том, что заказ был сохранен веб-сервисом.

Листинг 21.7. Содержимое файла `summary.tsx` из папки src

```
import React, { FunctionComponent } from "react";
import { NavLink, useParams } from "react-router-dom";

export const Summary : FunctionComponent = () => {
    const { id } = useParams();

    return <div className="m-2 text-center">
        <h2>Thanks!</h2>
        <p>Thanks for placing your order.</p>
        <p>Your order is #{" + id }</p>
        <p>We'll ship your goods as soon as possible.</p>
        <NavLink to="/products" className="btn btn-primary">OK</NavLink>
    </div>
}
```

Данный компонент использует хук `useParams`, который предоставляет доступ к параметрам, сопоставленным с маршрутом из текущего URL-пути. Этот хук позволяет компоненту получить идентификатор заказа, чтобы представить его пользователю. В листинге 21.12 определяется маршрут с параметром `id`.

21.3.2. Использование веб-сервиса

для выполнения заказов

Заказы создаются путем отправки выбранных пользователем товаров в веб-сервис, который был определен при создании проекта в главе 20. В листинге 21.8 для добавления поддержки веб-сервиса используются возможности, предоставляемые пакетом Redux Toolkit.

Листинг 21.8. Реализация веб-сервиса в файле `storeApis.ts` из папки src/data

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { Product, ProductSelection } from './entities';

const protocol = "http";
const hostname = "localhost";
const port = 4600;

const baseUrl = `${protocol}://${hostname}:${port}`;

export const productsApi = createApi({
    reducerPath: "products",
    baseQuery: fetchBaseQuery({baseUrl}),
    endpoints: (builder) => ({
        getProducts: builder.query<Product[], void>({
```

```

        query: () => "products"
    })
})
})

export const ordersApi = createApi({
    reducerPath: "orders",
    baseQuery: fetchBaseQuery({baseUrl}),
    endpoints: (build) => ({
        storeOrder: build.mutation<number, ProductSelection[]>({
            query(selections) {
                let orderData = {
                    lines: selections.map(ol => ({
                        productId: ol.product.id,
                        productName: ol.product.name,
                        quantity: ol.quantity
                    }))
                }
                return {
                    url: "orders",
                    method: "POST",
                    body: {orderData}
                }
            },
            transformResponse: ((response: {id: number}) => response.id)
        })
    })
}

export const { useGetProductsQuery } = productsApi;
export const { useStoreOrderMutation } = ordersApi;

```

Для реализации API используется функция `createApi` с мутацией, которая отправляет выбранные пользователем параметры в HTTP-запросе POST. Код из листинга 21.9 расширяет хранилище данных для включения в него поддержки дополнительных веб-сервисов.

Листинг 21.9. Расширение хранилища данных в файле `dataStore.ts` из папки `src/data`

```

import { configureStore } from "@reduxjs/toolkit";
import { reducer as selectionsReducer, addToOrder }
    from "./selectionSlice";
import { TypedUseSelectorHook, useDispatch, useSelector }
    from "react-redux";
import { ordersApi, productsApi, useGetProductsQuery,
    useStoreOrderMutation } from "./storeApis";

export const dataStore = configureStore({
    reducer: {
        "selections": selectionsReducer,
        [productsApi.reducerPath]: productsApi.reducer,
        [ordersApi.reducerPath]: ordersApi.reducer
    },

```

```

middleware: (getDefaultMiddleware) =>
  getDefaultMiddleware()
    .concat(productsApi.middleware)
    .concat(ordersApi.middleware)
};

export type AppDispatch = typeof dataStore.dispatch;
export type RootState = ReturnType<typeof dataStore.getState>;

export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;

export const reducers = {
  addToOrder, useStoreOrderMutation
}

export const queries = {
  useGetProductsQuery
}

```

21.3.3. Завершение работы над приложением

Для завершения работы над приложением необходимо внести несколько изменений. Первое — это расширить хранилище данных для того, чтобы была возможностьбросить выбор товаров после создания заказа (листинг 21.10).

Листинг 21.10. Добавление мутации в файл selectionSlice.ts из папки src/data

```

import { createSlice, PayloadAction } from "@reduxjs/toolkit";
import { Product, ProductSelection, ProductSelectionMutations }
  from "./entities";

const productSelectionSlice = createSlice({
  name: "selections",
  initialState: Array<ProductSelection>(),
  reducers: {
    addToOrder(selections: ProductSelection[],
      action: PayloadAction<[Product, number]>) {
      ProductSelectionMutations.addProduct(selections,
        action.payload[0], action.payload[1])
    },
    resetSelections(selections: ProductSelection[]) {

      selections.length = 0;
    }
  }
);

export const reducer = productSelectionSlice.reducer;
export const { addToOrder, resetSelections }

  = productSelectionSlice.actions

```

В листинге 21.11 импортируется и экспортируется новый редюсер, чтобы его можно было импортировать последовательно с другими функциями хранилища данных.

Листинг 21.11. Повторный экспорт функции в файле `dataStore.ts`
из папки `src/data`

```

import { configureStore } from "@reduxjs/toolkit";
import { reducer as selectionsReducer, addToOrder, resetSelections } from "./selectionSlice";
import { TypedUseSelectorHook, useDispatch, useSelector } from "react-redux";
import { ordersApi, productsApi, useGetProductsQuery, useStoreOrderMutation } from "./storeApis";

export const dataStore = configureStore({
  reducer: {
    "selections": selectionsReducer,
    [productsApi.reducerPath]: productsApi.reducer,
    [ordersApi.reducerPath]: ordersApi.reducer
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware()
      .concat(productsApi.middleware)
      .concat(ordersApi.middleware)
});
export type AppDispatch = typeof dataStore.dispatch;
export type RootState = ReturnType<typeof dataStore.getState>;

export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;

export const reducers = {
  addToOrder, useStoreOrderMutation, resetSelections
}

export const queries = {
  useGetProductsQuery
}

```

В листинге 21.12 добавлены новые элементы `Route` для отображения компонентов `OrderDetails` и `Summary`, что завершает конфигурацию маршрутизации для нашего приложения.

Листинг 21.12. Добавление оставшихся маршрутов в файл `App.tsx`
из папки `src`

```

import React, { FunctionComponent, useMemo } from 'react';
import { Product } from './data/entities';
import { ProductList } from './productList';
import { useAppDispatch, useAppSelector, reducers, queries } from "./data/dataStore";
import { Routes, Route, Navigate, useNavigate } from "react-router-dom";
import { Summary } from './summary';
import { OrderDetails } from './orderDetails';
import { resetSelections } from './data/selectionSlice';

export const App: FunctionComponent = () => {
  const selections = useAppSelector(state => state.selections);

```

```

const dispatch = useAppDispatch();

const { data } = queries.useGetProductsQuery();

const addToOrder = (p: Product, q: number) =>
    dispatch(reducers.addToOrder([p, q]));

const categories = useMemo<string[]>(() => {
    return [...new Set(data?.map(p => p.category))];
}, [data]);

const [ storeOrder ] = reducers.useStoreOrderMutation();
const navigate = useNavigate();
const submitCallback = () => {
    storeOrder(selections).unwrap().then(id => {
        dispatch(resetSelections());
        navigate('/summary/${id}');
    });
}

return (
    <div className="App">
        <Routes>
            <Route path="/products" element={(
                <ProductList products={ data ?? [] } 
                categories={categories} 
                selections={ selections } 
                addToOrder= { addToOrder } />
            ) />
            <Route path="/order" element={(
                <OrderDetails
                    selections={ selections }
                    submitCallback={() => submitCallback()} />
            ) />
            <Route path="/summary/:id" element={ <Summary /> } />
            <Route path="/" element={(
                <Navigate replace to="/products" />
            ) />
        </Routes>
    </div>
)
}
}

export default App;

```

Компонент маршрутизации `Route` для компонента `OrderDetails` предоставляет ему выбранные товары и функцию обратного вызова, которая отправляет эти товары на сервер и извлекает идентификатор, присвоенный сохраненному заказу. Выбор товаров сбрасывается, и браузеру предлагается перейти по пути `/summary` с сегментом, содержащим номер заказа.

Другой компонент `Route` отображает компонент `Summary`, а путь маршрута определяет параметр идентификатора, который считывается компонентом, чтобы отобразить его пользователю.

После сохранения изменений можно добавлять товары в заказ и отправлять его на веб-сервис (рис. 21.3).

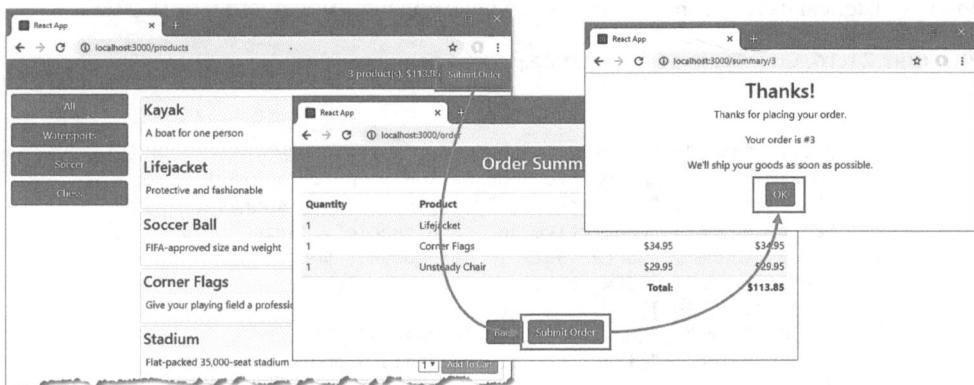


Рис. 21.3. Завершение работы над примером приложения

21.4. РАЗВЕРТЫВАНИЕ ПРИЛОЖЕНИЯ

Инструменты разработки React полагаются на сервер разработки Webpack, который не подходит для размещения производственного приложения, поскольку добавляет в генерируемые им пакеты JavaScript функцию автоматической перезагрузки. В этом разделе мы рассмотрим процесс подготовки приложения на React к развертыванию, который применим к любому веб-приложению, в том числе разработанному с использованием других фреймворков.

21.4.1. Добавление пакета производственного HTTP-сервера

Для производственной версии приложения требуется обычный HTTP-сервер доставки файлов HTML, CSS и JavaScript в браузер. Для нашего примера мы воспользуемся сервером Express, который применяется для других примеров в этой части книги и который является хорошим выбором для любого веб-приложения. С помощью комбинации клавиш **Control+C** остановите работу React и в папке `reactapp` выполните команду, показанную в листинге 21.13, для установки пакета `express`.

Вторая команда устанавливает пакет `connect-history-api-fallback`. Он полезен при развертывании приложений, использующих маршрутизацию URL-адресов, а также сопоставляет запросы URL-адресов, которые поддерживает приложение, с файлом `index.html`. Это гарантирует, что при перезагрузке браузера пользователь не получит ошибку `not found`.

Листинг 21.13. Добавление пакетов для развертывания

```
npm install --save-dev express@4.18.2
npm install --save-dev connect-history-api-fallback@2.0.0
```

21.4.2. Создание файла постоянных данных

Чтобы создать файл постоянных данных для веб-сервиса, добавьте в папку `reactapp` файл с именем `data.json` и поместите в него содержимое из листинга 21.14.

Листинг 21.14. Содержимое файла `data.json` из папки `reactapp`

```
{
  "products": [
    { "id": 1, "name": "Kayak", "category": "Watersports",
      "description": "A boat for one person", "price": 275 },
    { "id": 2, "name": "Lifejacket", "category": "Watersports",
      "description": "Protective and fashionable", "price": 48.95 },
    { "id": 3, "name": "Soccer Ball", "category": "Soccer",
      "description": "FIFA-approved size and weight",
      "price": 19.50 },
    { "id": 4, "name": "Corner Flags", "category": "Soccer",
      "description": "Give your playing field a professional touch",
      "price": 34.95 },
    { "id": 5, "name": "Stadium", "category": "Soccer",
      "description": "Flat-packed 35,000-seat stadium",
      "price": 79500 },
    { "id": 6, "name": "Thinking Cap", "category": "Chess",
      "description": "Improve brain efficiency by 75%",
      "price": 16 },
    { "id": 7, "name": "Unsteady Chair", "category": "Chess",
      "description": "Secretly give your opponent a disadvantage",
      "price": 29.95 },
    { "id": 8, "name": "Human Chess Board", "category": "Chess",
      "description": "A fun game for the family", "price": 75 },
    { "id": 9, "name": "Bling Bling King", "category": "Chess",
      "description": "Gold-plated, diamond-studded King",
      "price": 1200 }
  ],
  "orders": []
}
```

21.4.3. Создание сервера

Для создания сервера, который будет доставлять приложение и его данные в браузер, создайте в папке `reactapp` файл `server.js` и добавьте в него код, показанный в листинге 21.15.

Листинг 21.15. Содержимое файла `server.js` из папки `reactapp`

```
const express = require("express");
const jsonServer = require("json-server");
const history = require("connect-history-api-fallback");

const app = express();

const router = jsonServer.router("data.json");
app.use(jsonServer.bodyParser);
app.use("/api", (req, resp, next) => router(req, resp, next));
```

```
app.use(history());
app.use("/", express.static("build"));

const port = process.argv[3] || 4002;
app.listen(port, () => console.log('Running on port ${port}'));
```

Утверждения в файле `server.js` настраивают пакеты `express` и `json-server` таким образом, чтобы они использовали содержимое папки `build`, в которую в процессе сборки React будут помещены JavaScript-пакеты приложения и HTML-файл, указывающий браузеру на необходимость их загрузки. URL-адреса с префиксом `/api` будут обрабатываться веб-сервисом.

21.4.4. Использование относительных URL-адресов для запросов данных

Веб-сервис, предоставлявший приложению данные, работает параллельно с сервером разработки React. Чтобы подготовиться к отправке запросов на один порт, измените класс `HttpHandler` в соответствии с листингом 21.16.

Листинг 21.16. Использование относительных URL-адресов в файле `storeApis.ts` из папки `src/data`

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { Product, ProductSelection } from './entities';

//const protocol = "http";
//const hostname = "localhost";
//const port = 4600;

const baseUrl = "/api";

export const productsApi = createApi({
    reducerPath: "products",
    baseQuery: fetchBaseQuery({baseUrl}),
    endpoints: (builder) => ({
        getProducts: builder.query<Product[], void>({
            query: () => "products"
        })
    })
}

export const ordersApi = createApi({
    reducerPath: "orders",
    baseQuery: fetchBaseQuery({baseUrl}),
    endpoints: (build) => ({
        storeOrder: build.mutation<number, ProductSelection[]>({
            query(selections) {
                let orderData = {
                    lines: selections.map(ol => ({
                        productId: ol.product.id,
                    }))
                }
                return { data: orderData }
            }
        })
    })
}
```

```

        productName: ol.product.name,
        quantity: ol.quantity
    )))
}
return {
    url: "orders",
    method: "POST",
    body: {orderData}
}
},
transformResponse: ((response: {id : number}) => response.id)
)
)
})
}

export const { useGetProductsQuery } = productsApi;

export const { useStoreOrderMutation } = ordersApi;

```

21.4.5. Сборка приложения

Выполните команду из листинга 21.17 в папке `reactapp` для создания производственной сборки приложения, готового к эксплуатации.

Листинг 21.17. Создание производственного пакета

```
npm run build
```

В процессе сборки в папке `build` создается набор оптимизированных файлов. Данный процесс может занять несколько минут. В результате вы увидите следующее сообщение, где показано, какие файлы были включены в пакет:

```
Creating an optimized production build...
Compiled successfully.
```

`File sizes after gzip:`

```
81.69 kB build\static\js\main.61626b8e.js
28.01 kB build\static\css\main.ababef68.css
1.78 kB build\static\js\787.5480e000.chunk.js
```

```
The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.json.
```

```
The build folder is ready to be deployed.
You may serve it with a static server:
```

```
npm install -g serve
serve -s build
```

```
Find out more about deployment here:
https://cra.link/deployment
```

21.4.6. Тестирование производственной сборки

Чтобы убедиться, что процесс сборки прошел успешно и изменения конфигурации вступили в силу, выполните в папке `reactapp` команду из листинга 21.18.

Листинг 21.18. Запуск сервера

```
node server.js
```

После чего вы увидите следующий результат:

```
Running on port 4002
```

Откройте веб-браузер и перейдите по адресу `http://localhost:4002/products`, где будет показано приложение, как на рис. 21.4.



Рис. 21.4. Запуск финальной сборки

21.5. КОНТЕЙНЕРИЗАЦИЯ ПРИЛОЖЕНИЯ

В завершение этой главы мы создадим контейнер Docker для нашего примера приложения, чтобы его можно было развернуть в производственной среде. Если вы не установили Docker в главе 17, то вам необходимо сделать это сейчас, чтобы выполнить остальные примеры.

21.5.1. Подготовка приложения

Первым делом мы сформируем конфигурационный файл для NPM, который будет использоваться для загрузки дополнительных пакетов, необходимых приложению для работы в контейнере. Создайте в папке `reactapp` файл `deploy-package.json` с содержимым, показанным в листинге 21.19.

Листинг 21.19. Содержимое файла deploy-package.json из папки reactapp

```
{
  "name": "reactapp",
  "description": "React Web App",
  "repository": "https://github.com/manningbooks/essential-typescript-5",
  "license": "BSD",
  "devDependencies": {
    "express": "4.18.2",
    "json-server": "0.17.3",
    "connect-history-api-fallback": "2.0.0"
  }
}
```

В разделе `devDependencies` указываются пакеты, необходимые для запуска приложения в контейнере. Все пакеты, для которых в файлах кода приложения имеются операторы `import`, будут включены в пакет, созданный `webpack`, и перечислены в виде списка. Остальные поля описывают приложение и используются преимущественно для предотвращения появления предупреждений при создании контейнера.

21.5.2. Создание контейнера Docker

Чтобы определить контейнер, добавьте в папку `reactapp` файл с именем `Dockerfile` (без расширения), содержимое которого показано в листинге 21.20.

Листинг 21.20. Содержимое файла Dockerfile из папки reactapp

```
FROM node:18.14.0

RUN mkdir -p /usr/src/reactapp

COPY build /usr/src/reactapp/build/
COPY data.json /usr/src/reactapp/
COPY server.js /usr/src/reactapp/
COPY deploy-package.json /usr/src/reactapp/package.json

WORKDIR /usr/src/reactapp

RUN echo 'package-lock=false' >> .npmrc
RUN npm install

EXPOSE 4002

CMD ["node", "server.js"]
```

Содержимое `Dockerfile` использует базовый образ, настроенный на Node.js, который копирует в контейнер файлы, необходимые для запуска приложения, а также файл со списком пакетов, требуемых для развертывания.

Чтобы ускорить процесс контейнеризации, создайте в папке `reactapp` файл `.dockerignore` и поместите в него утверждение из листинга 21.21. Это говорит Docker игнорировать папку `node_modules`, которая не нужна в контейнере и отнимает много времени при обработке.

Листинг 21.21. Содержимое файла `.dockerignore` из папки `reactapp``node_modules`

Выполните команду из листинга 21.22 в папке `reactapp`, чтобы создать образ примера приложения вместе со всеми необходимыми пакетами.

Листинг 21.22. Создание образа Docker`docker build . -t reactapp -f Dockerfile`

Образ – это шаблон для контейнеров. По мере того как Docker обрабатывает инструкции в файле Docker, загружаются и устанавливаются пакеты NPM, а файлы конфигурации и кода скопируются в образ.

21.5.3. Запуск приложения

После формирования образа создайте и запустите новый контейнер с помощью команды, показанной в листинге 21.23.

Листинг 21.23. Запуск контейнера Docker`docker run -p 4002:4002 reactapp`

Вы можете протестировать приложение, открыв страницу `http://localhost:4002` в браузере, где будет отображен ответ, выданный веб-сервером, запущенным в контейнере (рис. 21.5).

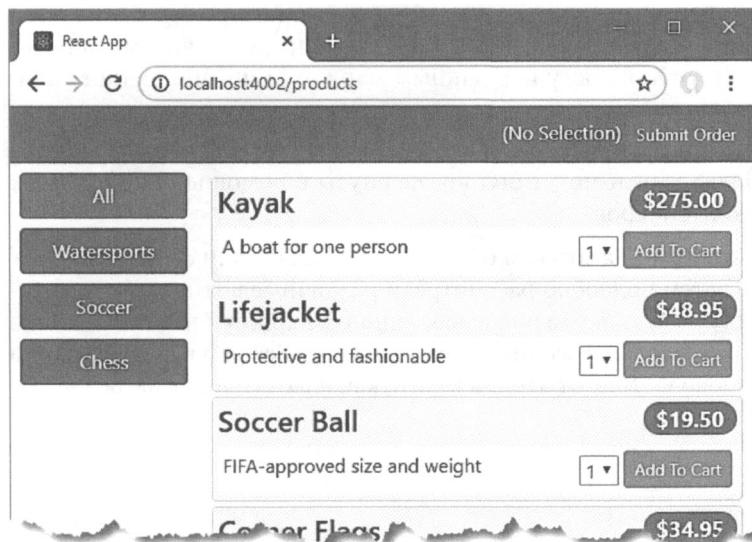


Рис. 21.5. Запуск контейнеризованного приложения

Для вывода списка запущенных контейнеров Docker на текущей машине выполните команду из листинга 21.24.

Листинг 21.24. Перечисление контейнеров

```
docker ps
```

Вы увидите примерно следующий список (для краткости я опустил некоторые поля):

CONTAINER ID	IMAGE	COMMAND	CREATED
82352eba95a2	reactapp	"docker-entry"	51 seconds ago

Используя значение в столбце CONTAINER ID, выполните команду, показанную в листинге 21.25, чтобы остановить работу контейнера.

Листинг 21.25. Остановка контейнера

```
docker stop 82352eba95a2
```

Теперь наше React-приложение готово к развертыванию на любой платформе, поддерживающей Docker.

РЕЗЮМЕ

В данной главе мы завершили работу над приложением на React, добавив поддержку маршрутизации URL-адресов и дополнительные компоненты. Как и в других примерах, приведенных в этой части книги, мы подготовили приложение к развертыванию и создали образ Docker, который можно легко развернуть.

- При маршрутизации отображаемые компоненты выбираются на основе текущего URL-адреса.
- Пакет маршрутизации предоставляет компоненты, определяющие маршруты, и хуки, используемые для доступа к данным маршрута и выполнения навигации.
- Как и любое другое приложение на TypeScript, проекты на React компилируются в чистый JavaScript и могут быть развернуты с помощью стандартных инструментов и контейнеров.

Вот и все, что я хотел рассказать вам о TypeScript. Мы начали с создания простого приложения, а затем подробно рассмотрели различные возможности, которые предоставляет TypeScript, в том числе как они применяются в системе типов JavaScript. Я желаю вам успехов в ваших проектах на TypeScript и надеюсь, что вы получили такое же удовольствие от чтения книги, как и я от ее написания.

Адам Фримен
Основы TypeScript

Перевед с английского А. Киселев

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Научный редактор	<i>В. Дмитрущенков</i>
Литературные редакторы	<i>А. Аверьянов</i>
Художественный редактор	<i>В. Мостапан</i>
Корректоры	<i>Е. Павлович, Н. Терех</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2024.
Наименование: книжная продукция.
Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 26.04.24. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 46,440. Тираж 700. Заказ 2463.

Отпечатано в полном соответствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
109316, г. Москва, Волгоградский проспект, д. 42,
корп. 5, эт. 1, пом. I, ком. 6.3-23Н

Мэтт Фрисби

JAVASCRIPT ДЛЯ ПРОФЕССИОНАЛЬНЫХ ВЕБ-РАЗРАБОТЧИКОВ.

4-е международное изд.



Самое полное руководство по современному JavaScript

Как максимально прокачать свои навыки и стать топовым JS-программистом? Четвертое издание «JavaScript для профессиональных веб-разработчиков» идеально подойдет тем, кто уже имеет базовые знания и опыт разработки на JavaScript. Автор сразу переходит к техническим деталям, которые сделают ваш код чистым и переведут вас с уровня рядового кодера на высоту продвинутого разработчика.

Рост мобильного трафика увеличивает потребность в адаптивном динамическом веб-дизайне, а изменения в JS-движках происходят постоянно, так что каждый веб-разработчик должен постоянно обновлять свои навыки работы с JavaScript.

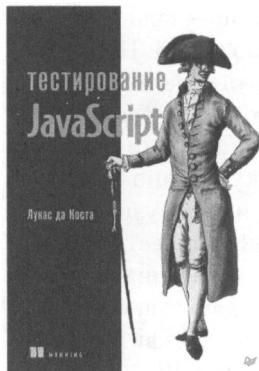
В книге вы найдете:

- Последнюю информацию о классах, промисах, `async/await`, прокси, итераторах, генераторах, символах, модулях и операторах `spread/rest`.
- Фундаментальные концепции веб-разработки, такие как DOM, BOM, события, формы, JSON, обработка ошибок и веб-анимация.
- Расширенные API-интерфейсы, такие как геолокация, service workers, `fetch`, атомизация, потоки, каналы сообщений и веб-криптография.
- Сотни рабочих примеров кода, которые ясно и кратко иллюстрируют концепции.



Лукас да Коста

ТЕСТИРОВАНИЕ JAVASCRIPT



Автоматизированное тестирование – залог стабильной разработки качественных приложений. Полноценное тестирование должно охватывать отдельные функции, проверять интеграцию разных частей вашего кода и обеспечивать корректность с точки зрения пользователя. Книга научит вас быстро и уверенно создавать надежное программное обеспечение. Вы узнаете, как реализовать план автоматизированного тестирования для JavaScript-приложений. В издании описываются стратегии тестирования, обсуждаются полезные инструменты и библиотеки, а также объясняется, как развивать культуру, ориентированную на качество. Вы исследуете подходы к тестированию как серверных, так и клиентских приложений, а также научитесь проверять свое программное обеспечение быстрее и надежнее.



Борис Черный

ПРОФЕССИОНАЛЬНЫЙ TYPESCRIPT. РАЗРАБОТКА МАСШТАБИРУЕМЫХ JAVASCRIPT-ПРИЛОЖЕНИЙ



Автоматизированное тестирование — залог стабильной разработки качественных приложений. Полноценное тестирование должно охватывать отдельные функции, проверять интеграцию разных частей вашего кода и обеспечивать корректность с точки зрения пользователя. Книга научит вас быстро и уверенно создавать надежное программное обеспечение. Вы узнаете, как реализовать план автоматизированного тестирования для JavaScript-приложений. В издании описываются стратегии тестирования, обсуждаются полезные инструменты и библиотеки, а также объясняется, как развивать культуру, ориентированную на качество. Вы исследуете подходы к тестированию как серверных, так и клиентских

приложений, а также научитесь проверять свое программное обеспечение быстрее и надежнее.



Основы TypeScript

Адам Фримен

TypeScript — популярная надстройка над JavaScript с поддержкой статической типизации, которая наверняка покажется знакомой программистам на C# или Java. TypeScript поможет вам сократить количество ошибок и повысить общее качество кода на JavaScript.

«Основы TypeScript» — это полностью обновленное третье издание классического бестселлера Адама Фримена. В нем освещены все возможности TypeScript 5, включая новые, такие как декораторы. Сначала вы узнаете, зачем и почему был создан язык TypeScript, а затем почти сразу перейдете к практическому применению статических типов. Ничего лишнего! Каждая глава посвящена навыкам, необходимым для написания потрясающих веб-приложений.

В этой книге

- Настройка средств разработки.
- Создание полностью типизированных функций и классов.
- Использование обобщенных типов, аннотаций типов и защиты типов.
- Создание и использование объявлений типов.

Для разработчиков на JavaScript.
Опыт работы с TypeScript не требуется.

Адам Фримен занимал руководящие должности в различных компаниях, в последнее время работал техническим директором и управляющим директором глобального банка. Написал 50 книг по программированию.

«Фантастическое введение в TypeScript. Доступное, понятное и непугающее».

— Джек Франклайн, Google

«Отличное учебное руководство, выполненное реалистичными примерами. Оно поможет вам освоить TypeScript и эффективно использовать его в ваших профессиональных и личных проектах».

— Ремо Янсен, Wolk Software

«Книга написана простым языком. Автор сразу задает хороший темп и приводит простые и понятные примеры, охватывающие все ключевые понятия. Здесь вы найдете все, что вам понадобится, чтобы начать работать».

— Стив Фентон, Octopus Deploy

«Эта книга дает ответы на некоторые давние вопросы, поднимает новые, а затем отвечает и на них. Незаменимое

руководство».

— Том Уэст, Cvent

 **ПИТЕР**®



WWW.PITER.COM
интернет-магазин

Заказ книг:
(812) 703-73-74
books@piter.com



 MANNING

ISBN: 978-5-4461-2215-8



9 785446 122158