

Оглавление

Предисловие	14
Благодарности	16
О книге	17
Целевая аудитория.....	17
Структура книги.....	17
О коде	19
Об авторе	19
Дискуссионный форум книги	19
Об иллюстрации на обложке	20
От издательства	20
Глава 1. Введение в типизацию.....	21
1.1. Для кого эта книга	22
1.2. Для чего существуют типы	22
1.2.1. Нули и единицы	23
1.2.2. Что такое типы и их системы	24
1.3. Преимущества систем типов.....	26
1.3.1. Корректность	26
1.3.2. Неизменяемость.....	28
1.3.3. Инкапсуляция	29
1.3.4. Компонуемость	31
1.3.5. Читабельность	33
1.4. Разновидности систем типов	34
1.4.1. Динамическая и статическая типизация.....	34
1.4.2. Слабая и сильная типизация.....	36
1.4.3. Вывод типов	37
1.5. В этой книге.....	38
Резюме	39

Глава 2. Базовые типы данных.....	40
2.1. Проектирование функций, не возвращающих значений.....	41
2.1.1. Пустой тип.....	41
2.1.2. Единичный тип	43
2.1.3. Упражнения	45
2.2. Булева логика и сокращенные схемы вычисления	45
2.2.1. Булевые выражения	46
2.2.2. Схемы сокращенного вычисления	46
2.2.3. Упражнение	48
2.3. Распространенные ловушки числовых типов данных	48
2.3.1. Целочисленные типы данных и переполнение	49
2.3.2. Типы с плавающей точкой и округление	53
2.3.3. Произвольно большие числа.....	56
2.3.4. Упражнения	56
2.4. Кодирование текста	57
2.4.1. Разбиение текста	57
2.4.2. Кодировки	58
2.4.3. Библиотеки кодирования	60
2.4.4. Упражнения	62
2.5. Создание структур данных на основе массивов и ссылок.....	62
2.5.1. Массивы фиксированной длины	62
2.5.2. Ссылки.....	64
2.5.3. Эффективная реализация списков	64
2.5.4. Бинарные деревья	67
2.5.5. Ассоциативные массивы.....	69
2.5.6. Соотношения выгод и потерь различных реализаций.....	70
2.5.7. Упражнение	71
Резюме	71
Ответы к упражнениям	72
Глава 3. Составные типы данных	73
3.1. Составные типы данных	74
3.1.1. Кортежи.....	74
3.1.2. Указание смыслового содержания.....	76
3.1.3. Сохранение инвариантов	77
3.1.4. Упражнение	80
3.2. Выражаем строгую дизъюнкцию с помощью типов данных.....	80
3.2.1. Перечисляемые типы	80
3.2.2. Опциональные типы данных	83
3.2.3. Результат или сообщение об ошибке	85
3.2.4. Вариантные типы данных.....	90
3.2.5. Упражнения	94

8 Оглавление

3.3. Паттерн проектирования «Посетитель».....	94
3.3.1. «Наивная» реализация	94
3.3.2. Использование паттерна «Посетитель».....	96
3.3.3. Посетитель-вариант.....	98
3.3.4. Упражнение	100
3.4. Алгебраические типы данных.....	100
3.4.1. Типы-произведения	101
3.4.2. Типы-суммы	101
3.4.3. Упражнения	102
Резюме	103
Ответы к упражнениям	103
Глава 4. Типобезопасность.....	105
4.1. Избегаем одержимости простыми типами данных, чтобы исключить неправильное толкование значений	106
4.1.1. Аппарат Mars Climate Orbiter	107
4.1.2. Антипаттерн одержимости простыми типами данных	109
4.1.3. Упражнение	110
4.2. Обеспечиваем соблюдение ограничений	110
4.2.1. Обеспечиваем соблюдение ограничений с помощью конструктора	111
4.2.2. Обеспечиваем соблюдение ограничений с помощью фабрики	112
4.2.3. Упражнение	113
4.3. Добавляем информацию о типе.....	113
4.3.1. Приведение типов.....	114
4.3.2. Отслеживание типов вне системы типов	115
4.3.3. Распространенные разновидности приведения типов.....	118
4.3.4. Упражнения	121
4.4. Скрываем и восстанавливаем информацию о типе	121
4.4.1. Неоднородные коллекции	122
4.4.2. СерIALIZАЦИЯ	125
4.4.3. Упражнения	128
Резюме	129
Ответы к упражнениям	129
Глава 5. Функциональные типы данных.....	131
5.1. Простой паттерн «Стратегия»	132
5.1.1. Функциональная стратегия	133
5.1.2. Типизация функций	135
5.1.3. Реализации паттерна «Стратегия»	135
5.1.4. Полноправные функции	136
5.1.5. Упражнения	137
5.2. Конечные автоматы без операторов switch.....	137
5.2.1. Предварительная версия книги	138
5.2.2. Конечные автоматы	140

5.2.3. Краткое резюме по реализации конечного автомата.....	146
5.2.4. Упражнения.....	147
5.3. Избегаем ресурсоемких вычислений с помощью отложенных значений.....	147
5.3.1. Лямбда-выражения.....	149
5.3.2. Упражнение.....	150
5.4. Использование операций map, filter и reduce	150
5.4.1. Операция map().....	151
5.4.2. Операция filter().....	153
5.4.3. Операция reduce()	155
5.4.4. Библиотечная поддержка.....	158
5.4.5. Упражнения.....	159
5.5. Функциональное программирование.....	159
Резюме	159
Ответы к упражнениям	160

Глава 6. Расширенные возможности применения функциональных типов данных	162
6.1. Простой паттерн проектирования «Декоратор».....	163
6.1.1. Функциональный декоратор	165
6.1.2. Реализации декоратора	166
6.1.3. Замыкания	167
6.1.4. Упражнение	168
6.2. Реализация счетчика.....	168
6.2.1. Объектно-ориентированный счетчик.....	169
6.2.2. Функциональный счетчик.....	170
6.2.3. Возобновляемый счетчик	171
6.2.4. Краткое резюме по реализациям счетчика.....	172
6.2.5. Упражнения	172
6.3. Асинхронное выполнение длительных операций	173
6.3.1. Синхронная реализация	173
6.3.2. Асинхронное выполнение: функции обратного вызова.....	174
6.3.3. Модели асинхронного выполнения.....	175
6.3.4. Краткое резюме по асинхронным функциям.....	179
6.3.5. Упражнения	180
6.4. Упрощаем асинхронный код	180
6.4.1. Сцепление промисов.....	182
6.4.2. Создание промисов	183
6.4.3. И еще о промисах	185
6.4.4. async/await.....	190
6.4.5. Краткое резюме по понятному асинхронному коду.....	191
6.4.6. Упражнения	192
Резюме	192
Ответы к упражнениям	193

Глава 7. Подтиповизация.....	195
7.1. Различаем схожие типы в TypeScript.....	196
7.1.1. Достоинства и недостатки номинальной и структурной подтиповизации	198
7.1.2. Моделирование номинальной подтиповизации в TypeScript.....	199
7.1.3. Упражнения.....	201
7.2. Присваиваем что угодно, присваиваем чему угодно	201
7.2.1. Безопасная десериализация.....	201
7.2.2. Значения на случай ошибки.....	206
7.2.3. Краткое резюме по высшим и низшим типам	209
7.2.4. Упражнения.....	209
7.3. Допустимые подстановки	209
7.3.1. Подтиповизация и типы-суммы.....	210
7.3.2. Подтиповизация и коллекции	212
7.3.3. Подтиповизация и возвращаемые типы функций.....	214
7.3.4. Подтиповизация и функциональные типы аргументов	216
7.3.5. Краткое резюме по вариантности	219
7.3.6. Упражнения.....	220
Резюме	221
Ответы к упражнениям	222
 Глава 8. Элементы объектно-ориентированного программирования.....	223
8.1. Описание контрактов с помощью интерфейсов	224
8.1.1. Упражнения	227
8.2. Наследование данных и поведения	228
8.2.1. Эмпирическое правило <i>is-a</i>	228
8.2.2. Моделирование иерархии	229
8.2.3. Параметризация поведения выражений.....	230
8.2.4. Упражнения	232
8.3. Композиция данных и поведения	232
8.3.1. Эмпирическое правило <i>has-a</i>	233
8.3.2. Композитные классы.....	234
8.3.3. Реализация паттерна проектирования «Адаптер»	236
8.3.4. Упражнения	237
8.4. Расширение данных и вариантов поведения	238
8.4.1. Расширение вариантов поведения с помощью композиции	239
8.4.2. Расширение поведения с помощью примесей.....	241
8.4.3. Примеси в TypeScript.....	242
8.4.4. Упражнение	244
8.5. Альтернативы чисто объектно-ориентированному коду	244
8.5.1. Типы-суммы	244
8.5.2. Функциональное программирование	247
8.5.3. Обобщенное программирование	248
Резюме	249
Ответы к упражнениям	249

Глава 9. Обобщенные структуры данных.....	251
9.1. Расцепление элементов функциональности.....	252
9.1.1. Повторно используемая тождественная функция.....	254
9.1.2. Тип данных Optional.....	255
9.1.3. Обобщенные типы данных.....	256
9.1.4. Упражнения.....	257
9.2. Обобщенное размещение данных.....	257
9.2.1. Обобщенные структуры данных	258
9.2.2. Что такое структура данных.....	259
9.2.3. Упражнения.....	260
9.3. Обход произвольной структуры данных.....	260
9.3.1. Использование итераторов	262
9.3.2. Делаем код итераций потоковым	266
9.3.3. Краткое резюме по итераторам.....	271
9.3.4. Упражнения	272
9.4. Потоковая обработка данных	273
9.4.1. Конвейеры обработки	273
9.4.2. Упражнения	275
Резюме	275
Ответы к упражнениям	276
Глава 10. Обобщенные алгоритмы и итераторы	279
10.1. Улучшенные операции map(), filter() и reduce()	280
10.1.1. Операция map()	280
10.1.2. Операция filter().....	281
10.1.3. Операция reduce()	282
10.1.4. Конвейер filter()/reduce()	283
10.1.5. Упражнения	283
10.2. Распространенные алгоритмы	284
10.2.1. Алгоритмы вместо циклов	285
10.2.2. Реализация текущего конвейера.....	285
10.2.3. Упражнения	289
10.3. Ограничение типов-параметров	289
10.3.1. Обобщенные структуры данных с ограничениями типа	290
10.3.2. Обобщенные алгоритмы с ограничениями типа	292
10.3.3. Упражнение	293
10.4. Эффективная реализация reverse и других алгоритмов с помощью итераторов	294
10.4.1. Стандартные блоки, из которых состоят итераторы	295
10.4.2. Удобный алгоритм find()	300
10.4.3. Эффективная реализация reverse().....	303
10.4.4. Эффективное извлечение элементов	306
10.4.5. Краткое резюме по итераторам.....	309
10.4.6. Упражнения	310

10.5. Адаптивные алгоритмы	310
10.5.1. Упражнение	312
Резюме	312
Ответы к упражнениям	313
Глава 11. Типы, относящиеся к более высокому роду, и не только.....	317
11.1. Еще более обобщенная версия алгоритма map.....	318
11.1.1. Обработка результатов и передача ошибок далее	321
11.1.2. Сочетаем и комбинируем функции.....	323
11.1.3. Функторы и типы, относящиеся к более высокому роду	324
11.1.4. Функторы для функций.....	327
11.1.5. Упражнение	329
11.2. Монады	329
11.2.1. Результат или ошибка.....	329
11.2.2. Различия между map() и bind()	334
11.2.3. Паттерн «Монада»	335
11.2.4. Монада продолжения.....	337
11.2.5. Монада списка	338
11.2.6. Прочие разновидности монад	340
11.2.7. Упражнение	341
11.3. Что изучать дальше.....	341
11.3.1. Функциональное программирование	341
11.3.2. Обобщенное программирование	342
11.3.3. Типы, относящиеся к более высокому роду, и теория категорий.....	342
11.3.4. Зависимые типы данных	343
11.3.5. Линейные типы данных	343
Резюме	344
Ответы к упражнениям	344
Приложение А. Установка TypeScript и исходный код.....	346
Онлайн	346
На локальной машине.....	346
Исходный код	346
«Самодельные» реализации	347
Приложение Б. Шпаргалка по TypeScript.....	348

Расширенные возможности применения функциональных типов данных

В этой главе

- Использование упрощенного паттерна проектирования «Декоратор».
- Реализация возобновляемого счетчика.
- Обработка длительных операций.
- Написание понятного асинхронного кода с помощью промисов и конструкции `async/await`.

В главе 5 мы рассмотрели основы функциональных типов данных и сценарии, ставшие возможными благодаря работе с функциями подобно любым другим значениям, то есть передаче их в качестве аргументов и возврате в виде результатов. Мы также рассмотрели несколько весьма многообещающих абстракций, реализующих распространенные паттерны обработки данных: `map()`, `filter()` и `reduce()`.

В этой главе мы продолжим обсуждение функциональных типов данных и их более продвинутых приложений. Начнем с паттерна проектирования «Декоратор» и его реализаций — традиционной и альтернативной. (Повторю: не волнуйтесь, если подзабыли его; я напомню, что к чему.) Вы познакомитесь с понятием «замыкание» (*closure*) и узнаете, как с его помощью реализовать простой счетчик. Затем рассмотрим другой способ реализации счетчика, на этот раз задействовав генератор — функцию, выдающую несколько результатов.

Далее мы поговорим об асинхронных операциях. Рассмотрим две основные модели асинхронного выполнения кода: потоки выполнения и циклы ожидания событий — и узнаем, как распланировать выполнение нескольких длительных операций. Мы начнем с функций обратного вызова, затем рассмотрим промисы и, наконец, поговорим о синтаксисе `async/await`, доступном сегодня в большинстве основных языков программирования.

Как мы увидим на последующих страницах, все обсуждаемые в этой главе темы возможны лишь благодаря использованию функций в качестве значений.

6.1. Простой паттерн проектирования «Декоратор»

«Декоратор» — это поведенческий паттерн проектирования программного обеспечения, который расширяет поведение объекта, не прибегая к модификации соответствующего класса. Декорированный объект способен на выполнение задач, выходящих за рамки возможностей его исходной реализации. Схема этого паттерна приведена на рис. 6.1.

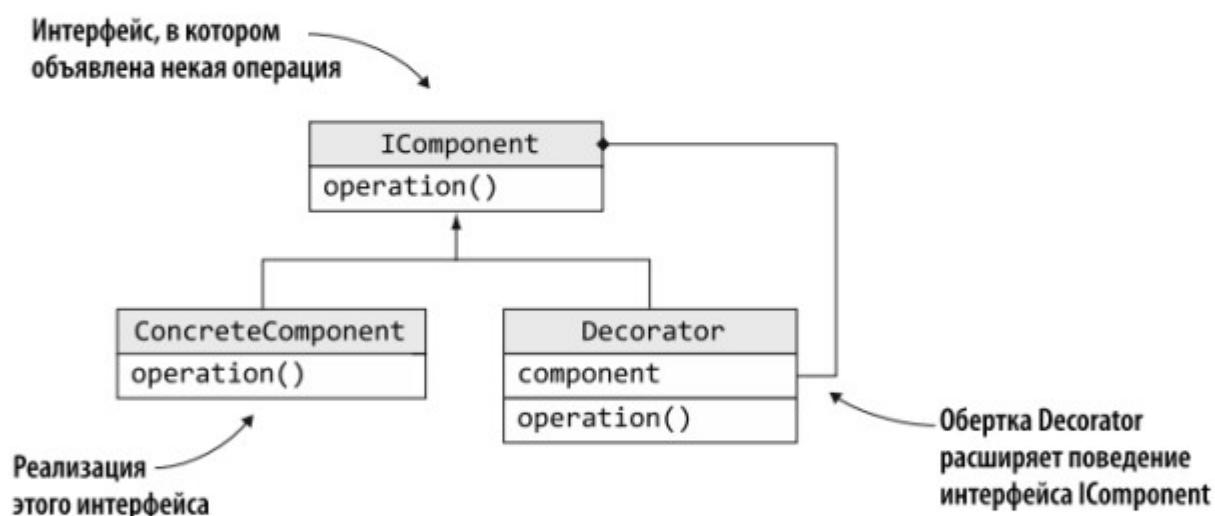


Рис. 6.1. Паттерн «Декоратор»: интерфейс `IComponent`, его конкретная реализация `ConcreteComponent` и `Decorator`, расширяющий `IComponent` дополнительным поведением

Для примера представим, что у нас есть интерфейс `IWidgetFactory`, в котором объявлен метод `Widget()`, возвращающий объект `Widget`. А в конкретной реализации `WidgetFactory` реализован метод для создания новых объектов `Widget`.

Допустим, что мы хотим повторно использовать `Widget` и вместо того, чтобы создавать каждый раз новый объект, хотели бы создать только один объект класса и всегда возвращать его (то есть реализовать одиночку). Не внося изменений в класс `WidgetFactory`, мы можем создать декоратор `SingletonDecorator` — обертку для `IWidgetFactory`, как показано в листинге 6.1, и расширить его поведение так, чтобы создавался лишь один объект `Widget` (рис. 6.2).

Листинг 6.1. Декоратор для IWidgetFactory

```

class Widget {}

interface IWidgetFactory {
    makeWidget(): Widget;
}

class WidgetFactory implements IWidgetFactory {
    public makeWidget(): Widget {
        return new Widget();
    }
}

class SingletonDecorator implements IWidgetFactory {
    private factory: IWidgetFactory;
    private instance: Widget | undefined = undefined;

    constructor(factory: IWidgetFactory) {
        this.factory = factory;
    }

    public makeWidget(): Widget {
        if (this.instance == undefined) {
            this.instance = this.factory.makeWidget(); ← WidgetFactory просто создает
        }                                     ← новый объект Widget
        return this.instance;                ← SingletonDecorator
    }                                     ← обертывает IWidgetFactory
}
}

```

Метод makeWidget реализует логику одиночки и гарантирует, что может быть создан только один экземпляр Widget

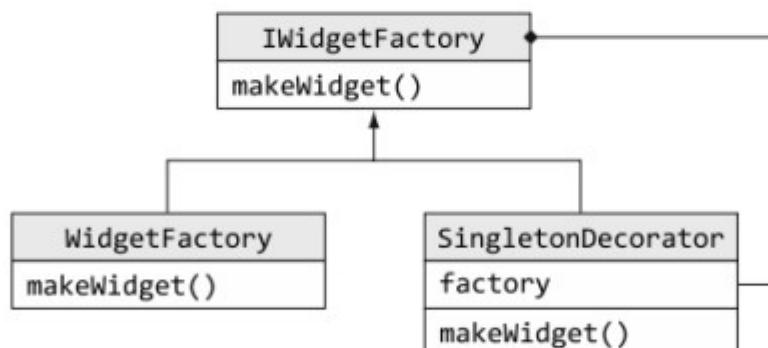


Рис. 6.2. Паттерн «Декоратор» для фабрики виджетов. IWidgetFactory — интерфейс, WidgetFactory — конкретная реализация, а класс SingletonDecorator добавляет в IWidgetFactory поведение одиночки

Преимущество этого паттерна заключается в поддержке *принципа единственной обязанности* (single-responsibility principle), который гласит: класс должен отвечать только за что-то одно. В данном случае класс `WidgetFactory` отвечает за создание виджетов, а `SingletonDecorator` — за поведение, соответствующее одиночке. Если нам потребуется несколько экземпляров класса, то можно воспользоваться непосредственно классом `WidgetFactory`. Если же один — классом `SingletonDecorator`.

6.1.1. Функциональный декоратор

Попробуем упростить эту реализацию опять-таки с помощью типизированных функций. Для начала избавимся от интерфейса `IWidgetFactory`, заменив его функциональным типом данных, описывающим функцию без аргументов, которая возвращает объект `Widget`: `() => Widget`.

Теперь мы можем заменить класс `WidgetFactory` простой функцией `makeWidget()`. Там, где раньше использовался интерфейс `IWidgetFactory` и передавался экземпляр `WidgetFactory`, теперь мы потребуем функции типа `() => Widget` и будем передавать туда `makeWidget()`, как показано в листинге 6.2.

Листинг 6.2. Функциональная фабрика виджетов

```
class Widget {}

type WidgetFactory = () => Widget; ← Функциональный тип данных
                           для фабрики виджетов

function makeWidget(): Widget {
    return new Widget();
} ← Тип функции makeWidget()
      соответствует типу WidgetFactory

function use10Widgets(factory: WidgetFactory) { ← Функция use10Widgets требует наличия
    for (let i = 0; i < 10; i++) {
        let widget = factory();
        /* ... */
    }
} ← Пример вызова: передаем функцию
use10Widgets(makeWidget); ← makeWidget в качестве аргумента
```

Для создания функциональной фабрики виджетов мы используем методику, очень близкую к паттерну проектирования «Стратегия» из главы 5: передаем функцию в качестве аргумента и вызываем ее при необходимости. Теперь посмотрим, как добавить сюда поведение одиночки.

Создаем новую функцию, `singletonDecorator()`, принимающую в качестве аргумента функцию типа `WidgetFactory` и возвращающую другую функцию типа `WidgetFactory`. Как вы помните из главы 5, лямбда-выражение — это функция без названия, которую можно возвращать из другой функции. В листинге 6.3 наш декоратор получает фабрику и с ее помощью создает новую функцию, отвечающую за поведение одиночки (рис. 6.3).

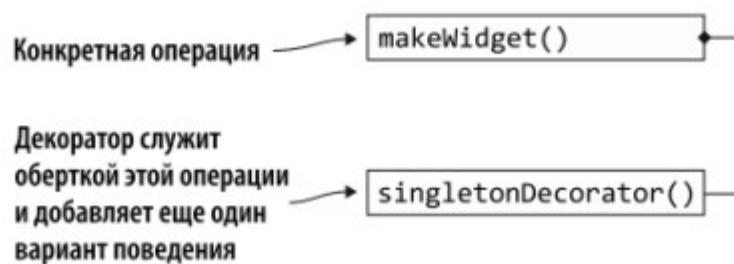


Рис. 6.3. Функциональный декоратор: теперь достаточно функций `makeWidget()` и `singletonDecorator()`

Листинг 6.3. Декоратор для функциональной фабрики виджетов

```

class Widget {}

type WidgetFactory = () => Widget;

function makeWidget(): Widget {
    return new Widget();
}

function singletonDecorator(factory: WidgetFactory): WidgetFactory {
    let instance: Widget | undefined = undefined;

    return (): Widget => {
        if (instance == undefined) {
            instance = factory();
        }
        return instance;
    };
}

function use10Widgets(factory: WidgetFactory) {
    for (let i = 0; i < 10; i++) {
        let widget = factory();
        /* ... */
    }
}

use10Widgets(singletonDecorator(makeWidget)); ←

```

Функция `singletonDecorator()` возвращает лямбда-выражение, реализующее поведение одиночки, используя заданную фабрику для создания объекта `Widget`

А поскольку функция `singletonDecorator()` возвращает `WidgetFactory`, ее можно передать в качестве аргумента функции `use10Widgets()`

Теперь вместо создания десяти объектов `Widget` функция `use10Widgets()` вызывает лямбда-выражение, повторно использующее один и тот же объект `Widget` для всех вызовов.

В этом коде количество компонентов уменьшается с интерфейса и двух классов — по одному методу каждый (конкретная операция и декоратор) — до двух функций.

6.1.2. Реализации декоратора

Как и в случае нашего паттерна «Стратегия», объектно-ориентированный и функциональный подход реализуют один и тот же паттерн проектирования «Декоратор». Объектно-ориентированная версия требует объявления интерфейса (`IWidgetFactory`), по крайней мере одной реализации этого интерфейса (`WidgetFactory`) и класса-декоратора, отвечающего за дополнительный вариант поведения (`SingletonDecorator`). При функциональной реализации же, напротив, просто объявляется тип для фабричной функции (`(()) => Widget`) и используются две функции: функция-фабрика (`makeWidget()`) и функция-декоратор (`singletonDecorator()`).

Стоит отметить, что в функциональном случае тип декоратора отличается от типа `makeWidget()`. У фабрики аргументов нет, она возвращает `Widget`, а декоратор принимает на входе фабрику виджетов и возвращает другую. Говоря иначе,

`singletonDecorator()` принимает в качестве аргумента функцию и возвращает ее в качестве результата. Это возможно только благодаря полноправности функций, то есть возможности работать с функциями точно так же, как и с прочими переменными, и использовать их в качестве аргументов и возвращаемых значений.

Эта более компактная реализация, ставшая доступной благодаря современным системам типов, вполне подходит для многих случаев. Более «многословное» объектно-ориентированное решение подходит для работы с несколькими функциями. Если в нашем интерфейсе объявлено несколько методов, то заменить их одним функциональным типом данных не получится.

6.1.3. Замыкания

Посмотрим более внимательно на реализацию `singletonDecorator()` в листинге 6.4. Возможно, вы обратили внимание на интересный нюанс: хоть функция возвращает лямбда-выражение, оно ссылается как на аргумент `factory`, так и на, казалось бы, локальную (по отношению к функции `singletonDecorator()`) переменную `instance`.

Листинг 6.4. Функция-декоратор

```
function singletonDecorator(factory: WidgetFactory): WidgetFactory {
    let instance: Widget | undefined = undefined;

    return (): Widget => {
        if (instance == undefined) {
            instance = factory();
        }

        return instance;
    };
}
```

И даже после возврата из функции `singletonDecorator()` переменная `instance` все равно существует, поскольку была «захвачена» лямбда-выражением. Это явление называется **лямбда-захватом** (*lambda capture*).

ЗАМЫКАНИЯ И ЛЯМБДА-ЗАХВАТЫ

Лямбда-захват представляет собой захват внешней переменной внутри лямбда-выражения. Такие захваты реализуются в языках программирования с помощью замыканий. Замыкание — это не просто функция: оно также фиксирует среду, в которой функция была создана, так что может сохранять состояние от вызова до вызова.

В нашем случае переменная `instance` в функции `singletonDecorator()` является частью такой среды, поэтому возвращенное лямбда-выражение по-прежнему сможет ссылаться на `instance` (рис. 6.4).

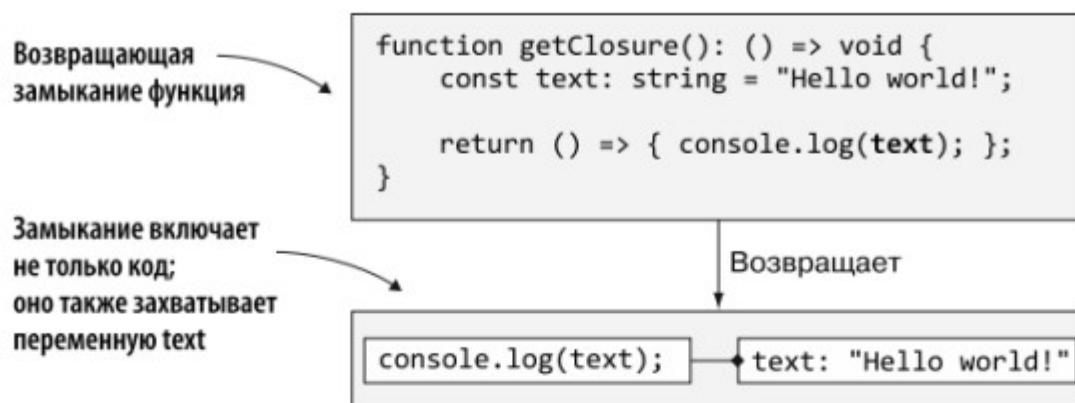


Рис. 6.4. Простая функция, возвращающая замыкание: лямбда-выражение, которое ссылается на локальную (по отношению к этой функции) переменную. Даже после возврата из функции `getClosure()` замыкание все равно ссылается на переменную, так что она существует дольше, чем функция, в которой появляется

Замыкания имеют смысл только при наличии функций высшего порядка. Если нельзя вернуть из одной функции другую, то нет и среды, которую можно было бы захватить. В этом случае все функции находятся в глобальной области видимости, которая и играет роль их среды. Они могут ссылаться на глобальные переменные.

Можно также сравнить замыкания с объектами. Объект — некое состояние с набором методов; *замыкание* — функция с неким захваченным состоянием. Рассмотрим еще один пример, в котором нам пригодятся замыкания, — реализацию счетчика.

6.1.4. Упражнение

Реализуйте функцию `loggingDecorator()`, принимающую в качестве аргумента другую функцию, `factory()`, которая не принимает аргументов и возвращает объект `Widget`. Декоратор должен вывести в консоль `"Widget created"`, прежде чем с помощью вызова заданной (переданной ему) функции вернуть объект `Widget`.

6.2. Реализация счетчика

Рассмотрим очень простой сценарий: создание счетчика, возвращающего последовательные числа, начиная с 1. Этот пример может показаться тривиальным, однако охватывает несколько возможных реализаций, которые можно применять любому сценарию генерации значений. Один из этих вариантов реализации — воспользоваться глобальной переменной и функцией, которая возвращает ее, после чего увеличивает ее значение на 1, как показано в листинге 6.5.

Данная реализация работает, но она не оптимальна. Во-первых, `n` — глобальная переменная, так что доступ к ней есть у кого угодно. Другой код может изменить ее значение незаметно для нас. Во-вторых, это реализация одного счетчика. А что, если нам понадобятся два счетчика, начинающихся с 1?