Оглавление

1 ■ Создание безопасных программ 31
2 ■ Функциональное программирование на Kotlin: обзор 46
3 ■ Программирование с функциями 81
4 ■ Рекурсия, сорекурсия и мемоизация 120
5 ■ Обработка данных с использованием списков 161
6 ■ Необязательные данные 197
7 ■ Обработка ошибок и исключений 222
8 ■ Дополнительные операции со списками 248
9 ■ Ленивые вычисления 282
10 ■ Обработка данных с использованием деревьев 323
11 ■ Решение задач с использованием усовершенствованных деревьев ... 363
12 ■ Функциональный ввод/вывод 392
13 ■ Общее изменяемое состояние и акторы 420
14 ■ Решение типичных проблем функциональным способом 448

Содержание

Оглавление 5
Предисловие 15
Благодарности 19
О книге 20
Об авторе 28
Об иллюстрации на обложке 29

1 ***Создание безопасных программ*** 31
1.1 Программные ловушки 33
1.1.1 Безопасная обработка эффектов 35
1.1.2 Увеличение безопасности программ за счет ссылочной прозрачности 36
1.2 Выгоды безопасного программирования 37
1.2.1 Использование подстановочной модели в рассуждениях о программе 39
1.2.2 Применение принципов соблюдения безопасности на простом примере 40
1.2.3 Максимальное использование абстракций 44
Итоги 45

2 ***Функциональное программирование на Kotlin: обзор*** 46
2.1 Поля и переменные в Kotlin 47
2.1.1 Тип можно опустить для простоты 47
2.1.2 Изменяемые поля 47
2.1.3 Отложенная инициализация 48
2.2 Классы и интерфейсы в Kotlin 49
2.2.1 Еще большее сокращение кода 51
2.2.2 Реализация интерфейса или расширение класса 51

2.2.3 Создание экземпляра класса ... 52
2.2.4 Перегрузка конструкторов ... 52
2.2.5 Создание методов equals и hashCode ... 53
2.2.6 Деструктуризация объектов данных ... 55
2.2.7 Реализация статических членов в Kotlin ... 55
2.2.8 Синглтоны ... 56
2.2.9 Предотвращение создания экземпляров служебных классов ... 56
2.3 В Kotlin нет элементарных типов ... 57
2.4 Два типа коллекций в Kotlin ... 57
2.5 Пакеты в Kotlin ... 59
2.6 Видимость в Kotlin ... 60
2.7 Функции в Kotlin ... 61
2.7.1 Объявление функций ... 61
2.7.2 Локальные функции ... 62
2.7.3 Переопределение функций ... 63
2.7.4 Функции-расширения ... 63
2.7.5 Лямбда-выражения ... 64
2.8 Пустое значение null в Kotlin ... 66
2.8.1 Приемы работы с типами, поддерживающими null ... 66
2.8.2 Оператор Элвис и значение по умолчанию ... 67
2.9 Поток выполнения программы и управляющие структуры ... 67
2.9.1 Условная конструкция ... 68
2.9.2 Использование конструкций с несколькими условиями ... 69
2.9.3 Циклы ... 70
2.9.4 Какие проблемы может вызывать поддержка вариантности? ... 76
2.9.5 Когда использовать объявления ковариантности и контрвариантности ... 77
2.9.6 Объявление вариантности в точке определения и точке использования ... 78
Итоги ... 79

3 *Программирование с функциями* ... 81
3.1 Что такое функция? ... 82
3.1.1 Отношения между двумя областями функций ... 83
3.1.2 Обратные функции в Kotlin ... 84
3.1.3 Частичные функции ... 85
3.1.4 Композиция функций ... 86
3.1.5 Функции нескольких аргументов ... 86
3.1.6 Каррирование функций ... 87
3.1.7 Частично-примененные функции ... 87
3.1.8 Функции не имеют эффектов ... 88
3.2 Функции в Kotlin ... 89

3.2.1 Функции как данные ... 89
3.2.2 Данные как функции ... 89
3.2.3 Конструкторы объектов как функции ... 89
3.2.4 Использование функций fun в Kotlin ... 90
3.2.5 Объектная и функциональная нотация ... 93
3.2.6 Использование функций как значений ... 94
3.2.7 Ссылки на функции ... 96
3.2.8 Композиция функций fun ... 97
3.2.9 Повторное использование функций ... 98
3.3 Дополнительные особенности функций ... 99
3.3.1 Функции с несколькими аргументами? ... 99
3.3.2 Применение каррированных функций ... 100
3.3.3 Реализация функций высшего порядка ... 100
3.3.4 Полиморфные функции высшего порядка ... 102
3.3.5 Анонимные функции ... 104
3.3.6 Локальные функции ... 106
3.3.7 Замыкания ... 106
3.3.8 Частичное применение функций и автоматическое каррирование ... 107
3.3.9 Перестановка аргументов частично примененных функций ... 112
Итоги ... 119

4 *Рекурсия, сорекурсия и мемоизация* ... 120
4.1 Рекурсия и сорекурсия ... 121
4.1.1 Реализация сорекурсии ... 121
4.1.2 Реализация рекурсии ... 123
4.1.3 Различия между рекурсивными и сорекурсивными функциями ... 124
4.1.4 Выбор между рекурсией и сорекурсией ... 125
4.2 Удаление хвостового вызова ... 127
4.2.1 Использование механизма удаления хвостового вызова ... 128
4.2.2 Преобразование циклов в хвостовую рекурсию ... 128
4.2.3 Рекурсивные функции-значения ... 132
4.3 Рекурсивные функции и списки ... 135
4.3.1 Дважды рекурсивные функции ... 137
4.3.2 Абстракция рекурсии в списках ... 140
4.3.3 Обращение списка ... 143
4.3.4 Сорекурсивные списки ... 145
4.3.5 Опасность строгости ... 149
4.4 Мемоизация ... 149
4.4.1 Мемоизация в программировании на основе циклов ... 149
4.4.2 Мемоизация рекурсивных функций ... 150
4.4.3 Неявная мемоизация ... 152
4.4.4 Автоматическая мемоизация ... 154
4.4.5 Мемоизация функций нескольких аргументов ... 157

4.5 Являются ли мемоизованные функции чистыми? ... 159
Итоги ... 159

5 *Обработка данных с использованием списков* ... 161
5.1 Классификация коллекций данных ... 162
5.2 Разные типы списков ... 162
5.3 Производительность списков ... 164
5.3.1 Время в обмен на объем памяти и сложность ... 165
5.3.2 Отказ от изменения на месте ... 165
5.4 Какие виды списков доступны в Kotlin? ... 167
5.4.1 Использование постоянных структур данных ... 168
5.4.2 Реализация неизменяемого, постоянного, односвязного списка ... 168
5.5 Совместное использование данных в операциях со списками ... 172
5.6 Дополнительные операции со списками ... 174
5.6.1 Преимущества объектной нотации ... 175
5.6.2 Объединение списков ... 178
5.6.3 Удаление элементов с конца списка ... 180
5.6.4 Использование рекурсии для свертки списков с помощью функций высшего порядка ... 181
5.6.5 Вариантность ... 182
5.6.6 Безопасная рекурсивная версия foldRight ... 191
5.6.7 Отображение и фильтрация списков ... 193
Итоги ... 196

6 *Необязательные данные* ... 197
6.1 Проблемы с пустым указателем ... 198
6.2 Как пустые ссылки обрабатываются в Kotlin ... 201
6.3 Альтернативы пустым ссылкам ... 202
6.4 Тип Option ... 205
6.4.1 Извлечение значения из Option ... 207
6.4.2 Применение функций к необязательным значениям ... 209
6.4.3 Композиция функций с типом Option ... 210
6.4.4 Примеры использования Option ... 212
6.4.5 Другие способы комбинирования типа Options ... 215
6.4.6 Комбинирование List и Option ... 218
6.4.7 Когда и как использовать тип Option ... 220
Итоги ... 221

7 *Обработка ошибок и исключений* ... 222
7.1 Проблемы с отсутствующими данными ... 223
7.2 Тип Either ... 224
7.3 Тип Result ... 227

7.4 Приемы использования типа Result 230
7.5 Дополнительные способы использования Result 236
7.5.1 Применение предикатов 236
7.6 Преобразование ошибок 238
7.7 Дополнительные фабричные функции 239
7.8 Применение эффектов 240
7.9 Дополнительные способы комбинирования с типом Result 243
Итоги 247

8 ***Дополнительные операции со списками*** 248
8.1 Проблемы функции length 249
8.2 Проблема производительности 249
8.3 Преимущества мемоизации 250
8.3.1 Недостатки мемоизации 250
8.3.2 Оценка увеличения производительности 252
8.4 Комбинирование List и Result 253
8.4.1 Списки, возвращающие Result 253
8.4.2 Преобразование List<Result> в Result<List> 255
8.5 Абстракции операций со списками 257
8.5.1 Упаковка и распаковка списков 258
8.5.2 Доступ к элементам по индексам 261
8.5.3 Разбиение списков 266
8.5.4 Поиск подсписков 270
8.5.5 Разные функции для работы со списками 271
8.6 Автоматическое распараллеливание операций со списками 276
8.6.1 Не все вычисления могут выполняться параллельно 276
8.6.2 Деление списка на подсписки 276
8.6.3 Параллельная обработка подсписков 278
Итоги 280

9 ***Ленивые вычисления*** 282
9.1 Строгий и ленивый подходы 283
9.2 Строгие вычисления в Kotlin 284
9.3 Ленивые вычисления в Kotlin 286
9.4 Реализация ленивых вычислений 288
9.4.1 Комбинирование ленивых значений 290
9.4.2 Преобразование обычных функций в ленивые 294
9.4.3 Отображение ленивых значений 296
9.4.4 Комбинирование типов Lazy и List 298
9.4.5 Обработка исключений 299
9.5 Другие способы комбинирования ленивых вычислений 302

9.5.1 Ленивое применение эффектов 302
9.5.2 Вычисления, невозможные без ленивых значений 304
9.5.3 Создание ленивого списка 305
9.6 Работа с потоками 308
9.6.1 Свертка потоков 314
9.6.2 Трассировка вычислений и применение функций 317
9.6.3 Использование потоков для решения конкретных задач 319
Итоги 322

10 *Обработка данных с использованием деревьев* 323
10.1 Бинарное дерево 324
10.2 Сбалансированные и несбалансированные деревья 325
10.3 Размер, высота и глубина дерева 325
10.4 Пустые деревья и рекурсивное определение 326
10.5 Лиственные деревья 327
10.6 Упорядоченные бинарные деревья, или бинарные деревья поиска 327
10.7 Порядок вставки и структура дерева 329
10.8 Рекурсивный и нерекурсивный обход дерева 330
10.8.1 Рекурсивный обход дерева 330
10.8.2 Нерекурсивный обход дерева 332
10.9 Реализация бинарного дерева поиска 333
10.9.1 Деревья и вариантность 334
10.9.2 Об абстрактных функциях в классе Tree 336
10.9.3 Перегрузка операторов 336
10.9.4 Рекурсия в деревьях 336
10.9.5 Удаление элементов из дерева 340
10.9.6 Слияние произвольных деревьев 341
10.10 О свертке деревьев 347
10.10.1 Свертка с двумя функциями 348
10.10.2 Свертка с одной функцией 350
10.10.3 Выбор реализации свертки 350
10.11 О преобразовании элементов деревьев 353
10.12 О балансировке деревьев 354
10.12.1 Вращение деревьев 354
10.12.2 Алгоритм Дея/Стоута/Уоррен 357
10.12.3 Самобалансирующиеся деревья 361
Итоги 362

11 *Решение задач с использованием усовершенствованных деревьев* 363
11.1 Улучшение производительности и безопасности деревьев добавлением самобалансировки 364
11.1.1 Структура красно-черных деревьев 365

11.1.2 Добавление элемента в красно-черное дерево 367
11.1.3 Удаление элементов из красно-черного дерева 373
11.2 Практические примеры использования красно-черных деревьев: ассоциативные массивы 373
11.2.1 Реализация Map 373
11.2.2 Расширение ассоциативных массивов 376
11.2.3 Использование ключей, не поддерживающих сравнение 377
11.3 Реализация функциональной приоритетной очереди 380
11.3.1 Протоколы доступа к приоритетной очереди 380
11.3.2 Варианты использования приоритетных очередей 380
11.3.3 Требования к реализации 381
11.3.4 Левосторонняя куча 381
11.3.5 Реализация левосторонней кучи 382
11.3.6 Реализация интерфейса, характерного для очередей 385
11.4 Элементы и сортированные списки 386
11.5 Приоритетная очередь для несопоставимых элементов 388
Итоги 391

12 *Функциональный ввод/вывод* 392
12.1. Что означает «эффект внутри контекста»? 393
12.1.1 Обработка эффектов 394
12.1.2 Реализация эффектов 394
12.2 Чтение данных 397
12.2.1 Чтение данных с клавиатуры 398
12.2.2 Чтение из файла 402
12.3 Тестирование программ с вводом 404
12.4 Полностью функциональный ввод/вывод 405
12.4.1 Как сделать ввод/вывод полностью функциональным 405
12.4.2 Реализация чисто функционального ввода/вывода 406
12.4.3 Комбинирование ввода/вывода 407
12.4.4 Обработка ввода с IO 409
12.4.5 Расширение типа IO 411
12.4.6 Добавление в IO защиты от переполнения стека 414
Итоги 419

13 *Общее изменяемое состояние и акторы* 420
13.1 Модель акторов 422
13.1.1 Асинхронный обмен сообщениями 422
13.1.2 Параллельное выполнение 422
13.1.3 Управление изменением состояния актора 423
13.2 Реализация инфраструктуры акторов 424
13.2.1 Обзор ограничений 425
13.2.2 Интерфейсы инфраструктуры акторов 425
13.3 Реализация AbstractActor 427

13.4 Включение акторов в работу ... 428
13.4.1 Реализация примера игры в пинг-понг ... 429
13.4.2 Параллельное выполнение вычислений ... 431
13.4.3 Переупорядочение результатов ... 437
13.4.4 Оптимизация производительности ... 440
Итоги ... 447

14 *Решение типичных проблем функциональным способом* ... 448
14.1 Утверждения и проверка данных ... 449
14.2 Повторный вызов функций и эффектов ... 453
14.3 Чтение свойств из файла ... 456
14.3.1 Загрузка файла со свойствами ... 457
14.3.2 Чтение свойств как строк ... 458
14.3.3 Вывод более информативных сообщений об ошибках ... 459
14.3.4 Чтение свойств как списков ... 462
14.3.5 Чтение значений перечислений ... 463
14.3.6 Чтение свойств произвольных типов ... 464
14.4 Преобразование императивной программы: чтение файлов XML ... 467
14.4.1 Шаг 1: императивное решение ... 468
14.4.2 Шаг 2: превращаем императивную программу в функциональную ... 470
14.4.3 Шаг 3: делаем программу еще более функциональной ... 473
14.4.4 Шаг 4: исправление проблемы с аргументами одного типа ... 477
14.4.5 Шаг 5: передача функции обработки элемента в параметре ... 478
14.4.6 Шаг 6: обработка ошибок в именах элементов ... 479
14.4.7 Шаг 7: дополнительные улучшения в прежде императивном коде ... 481
Итоги ... 483

Приложение А. Смешивание кода на Kotlin и Java ... 484
Создание и управление смешанными проектами ... 485
Создание простого проекта в GRADLE ... 485
Импортирование Gradle-проекта в IntelliJ ... 487
Добавление зависимостей в проект ... 488
Создание проектов с несколькими модулями ... 488
Добавление зависимостей в проект с несколькими модулями ... 489
Вызов Java-методов из Kotlin ... 490
Использование примитивов Java ... 490
Использование числовых типов-объектов Java ... 491
Быстрый отказ со значениями null ... 492
Использование строковых типов Kotlin и Java ... 492

Преобразование других типов ... 493
Вызов Java-методов с переменным числом параметров ... 494
Управление поддержкой null в Java ... 494
Доступ к свойствам в JAVA с зарезервированными именами ... 497
Вызов контролируемых исключений ... 497
SAM-интерфейсы ... 498
Вызов Kotlin-функций из Java ... 498
Преобразование свойств Kotlin ... 498
Использование общедоступных полей Kotlin ... 499
Статические поля ... 499
Вызов функций Kotlin из методов Java ... 500
Преобразование типов Kotlin в типы Java ... 503
Типы функций ... 504
Характерные проблемы смешанных проектов на Kotlin/Java ... 504
Итоги ... 505

Приложение B. Тестирование на основе свойств ... 507
Зачем нужно тестирование на основе свойств? ... 508
Интерфейс ... 509
Тест ... 509
Что такое тестирование на основе свойств? ... 510
Абстракция и тестирование на основе свойств ... 511
Зависимости для модульного тестирования на основе свойств ... 513
Разработка тестов на основе свойств ... 514
Создание своих генераторов ... 517
Использование своих генераторов ... 518
Упрощение кода дальнейшим абстрагированием ... 522
Итоги ... 524

Предметный указатель ... 526

Предисловие

Kotlin появился еще в 2011 году, но по-прежнему остается одним из самых новых языков в экосистеме Java. С тех пор появилась версия Kotlin, работающая на виртуальной машине JavaScript, а также версия, выполняющая компиляцию в машинный код. Это делает Kotlin гораздо более универсальным языком, чем Java, хотя между этими версиями есть большие различия, потому что версия для виртуальной машины Java опирается на стандартную библиотеку Java, недоступную в двух других версиях Kotlin. Сотрудники компании JetBrains, где создан язык Kotlin, прикладывают все силы, чтобы вывести все версии на один уровень, но, как вы понимаете, это требует времени.

Версия для JVM (Java Virtual Machine – виртуальной машины Java), безусловно, является наиболее широко используемой, и она получила дополнительный импульс к развитию, когда в Google объявили Kotlin одним из официальных языков разработки приложений для Android. Основная причина такого решения Google состоит в том, что последняя доступная в Android версия Java – это Java 6, тогда как Kotlin предлагает большинство особенностей Java 11 и многое другое. Kotlin также был объявлен официальным языком сценариев сборки в Gradle вместо Groovy, что позволяет использовать один и тот же язык и для сборки программ, и для самих собираемых программ.

В первую очередь язык Kotlin ориентирован на программистов на Java. Вполне возможно, что в будущем они будут изучать Kotlin как основной язык. Но пока основная масса программистов приходит в Kotlin из Java.

У каждого языка свой путь, определяемый некоторыми фундаментальными понятиями. Язык Java создавался на основе нескольких мощных идей. Он должен работать везде, т. е. в любой среде, где доступна JVM. Главный девиз: «Пиши один раз, запускай где угодно». Хотя некоторые могут утверждать обратное, но в целом Java соответствует этому девизу. Более того, теперь можно запускать почти везде не только программы на Java, но и программы на других языках, скомпилированные для JVM. Kotlin – один из таких языков.

Еще одна основополагающая идея Java – никакие нововведения никогда не станут причиной неработоспособности существующего кода. Хотя эта идея соблюдалась не всегда, но все же разработчики языка старались следовать ей. Конечно, это не всегда хорошо. Основным следствием является невозможность внесения в Java многих улучшений, имеющихся в других языках, потому что они могут привести к нарушению совместимости. Любая программа, скомпилированная с предыдущей версией Java, должна оставаться работоспособной в более новых версиях без перекомпиляции. Является ли это полезным или нет, это спорный вопрос, но в результате стремление максимально сохранить обратную совместимость постоянно играет против развития Java.

Также предполагалось, что Java сделает программы более безопасными за счет использования контролируемых исключений, что вынуждает программистов принимать эти исключения во внимание. Для многих это оказалось тяжким бременем, ведущим к практике постоянного преобразования контролируемых исключений в неконтролируемые.

Хотя Java является объектно-ориентированным языком, он должен быть настолько же быстрым, как и большинство языков, используемых для вычислительных задач. В свое время разработчики языка решили, что Java только выиграет, если, помимо объектов, представляющих числа и логические значения, в языке будут поддерживаться элементарные числовые типы, позволяющие выполнять вычисления намного быстрее. Из-за этого отсутствует возможность помещать значения примитивных типов в коллекции, такие как списки, множества и ассоциативные массивы. А когда были добавлены потоки данных (streams), разработчики языка решили создать специальные версии для примитивов, но не для всех, а только для наиболее часто используемых. Если вы пользуетесь некоторыми из неподдерживаемых примитивов, значит, вам не повезло.

То же самое произошло с функциями. В Java 8 появилась поддержка обобщенных функций, но обобщение возможно только для объектов. Поэтому для обработки целых, длинных целых, вещественных чисел двойной точности и логических значений были разработаны специализированные функции. (И снова, к сожалению, поддержка добавлена не для всех элементарных типов – для работы с однобайтовыми и короткими целыми, а также для вещественных чисел одинарной точности нет специализированных функций.) Что еще хуже, потребовались дополнительные функции для преобразования из одного элементарного типа в другой или из элементарных типов в объекты и обратно.

Язык Java был разработан более 20 лет назад. С тех пор многое изменилось, но большинство этих изменений нельзя было перенести в Java, потому что это нарушило бы совместимость. Некоторые изменения все же вносились, и совместимость сохранилась, но за счет удобства использования.

Для устранения этих ограничений было создано много новых языков, таких как Groovy, Scala и Clojure. Они в определенной степени совместимы с Java и позволяют применять существующие библиотеки Java,

а программисты на Java могут использовать библиотеки, написанные на этих языках.

Kotlin пошел другим путем. Он гораздо сильнее интегрирован с Java, что позволяет без проблем смешивать исходный код на Kotlin и Java в одном проекте! В отличие от других языков для JVM Kotlin не выглядит сильно отличающимся от Java (хотя разница, конечно же, имеется). Он больше похож на язык, которым должен стать язык Java. Некоторые даже говорят, что Kotlin – это Java, созданный правильно, потому что решает большинство проблем, характерных для Java. (Однако Kotlin пока не предлагает решений, связанных с ограничениями, свойственными JVM.)

Но, что еще более важно, Kotlin разрабатывался так, чтобы быть намного более восприимчивым ко многим новым приемам, появляющимся в функциональном программировании. В Kotlin есть изменяемые и неизменяемые ссылки, но предпочтение отдается неизменяемым. Kotlin также поддерживает большую часть абстракций функционального программирования, которые позволяют избегать управляющих структур (хотя он имеет традиционные управляющие структуры, помогающие упростить переход от традиционных языков).

Еще одна замечательная особенность Kotlin – он уменьшает потребность в шаблонном коде, позволяя сократить его до минимума. На Kotlin можно написать класс с необязательными свойствами (обладающий также функциями `equals`, `hashCode`, `toString` и `copy`) в одной строке кода, тогда как для объявления эквивалентного класса на Java потребуется около тридцати строк (включая методы свойств и перегруженные конструкторы).

Да, есть другие языки программирования, разработанные для преодоления ограничений Java в среде JVM, но Kotlin отличается тем, что прекрасно интегрируется с Java-программами на уровне исходного кода. Вы можете смешивать исходные файлы на Java и Kotlin в проектах и использовать единую цепочку сборки. Это меняет правила игры, особенно в отношении командного программирования, потому что использование Kotlin в среде Java – не более сложная задача, чем использование любой сторонней библиотеки. Это обеспечивает максимально плавный переход с Java на новый язык и позволяет писать программы, которые:

- безопаснее;
- проще в разработке, тестировании и сопровождении;
- более масштабируемые.

Я полагаю, что многие читатели – программисты на Java – рано или поздно осознают необходимость поиска новых решений своих повседневных проблем. Возможно, у вас уже возник вопрос, почему вы должны использовать Kotlin. Нет ли других языков в экосистеме Java, которые позволят легко применять безопасные методы программирования?

Конечно, есть, и одним из самых известных является Scala. Scala – очень хорошая альтернатива Java, но у Kotlin есть нечто большее. Scala может

взаимодействовать с Java на уровне библиотек, т. е. программы на Java могут использовать библиотеки Scala (объекты и функции), а программы на Scala могут использовать библиотеки Java (объекты и методы). Но программы на Scala и Java должны создаваться как отдельные проекты или хотя бы как отдельные модули, тогда как классы Kotlin и Java можно смешивать внутри одного модуля.

Прочитайте эту книгу, чтобы узнать больше о Kotlin.

Благодарности

Я хотел бы поблагодарить всех, кто участвовал в создании этой книги. Прежде всего, большое спасибо моему редактору Марине Майклз (Marina Michaels). Мне было очень приятно работать с Вами. Также спасибо моему научному редактору Александару Драгосавлевичу (Aleksandar Dragosavljević).

Большое спасибо также Джоэлю Котарски (Joel Kotarski), Джошуа Уайту (Joshua White) и Риккардо Терреллу (Riccardo Terrell) – техническим редакторам, Алессандро Кампейсу (Alessandro Campeis) и Бренту Уотсону (Brent Watson) – корректорам, которые помогли мне сделать эту книгу намного лучше. Спасибо всем рецензентам, читателям, участвующим в программе MEAP, и всем, кто прислал свои отзывы и комментарии, спасибо вам! Эта книга не получилась бы такой, какая она есть, без вашей помощи. Также я хотел бы поблагодарить людей, которые нашли время, чтобы просмотреть и прокомментировать книгу: Алексея Слайковского (Aleksei Slaikovskii), Алессандро Кампейса (Alessandro Campeis), Энди Кирша (Andy Kirsch), Бенджамина Голдберга (Benjamin Goldberg), Бриджера Хауэлла (Bridger Howell), Конора Редмонда (Conor Redmond), Дилана Макнейми (Dylan McNamee), Эммануэль Мединy Лопес (Emmanuel Medina López), Фабио Фальси Родригес (Fabio Falci Rodrigues), Федерико Кирхейса (Federico Kircheis), Герго Михай Надя (Gergő Mihály Nagy), Грегора Раймана (Gregor Raýman), Джейсона Ли (Jason Lee), Жан-Франсуа Морина (Jean-François Morin), Кента Р. Спилнера (Kent R. Spillner), Линн Нортроп (Leanne Northrop), Марка Элстона (Mark Elston), Мэтью Халверсона (Matthew Halverson), Мэтью Проктора (Matthew Proctor), Нуно Александра (Nuno Alexandre), Рафаэля Вентальо (Raffaella Ventaglio), Рональда Харинга (Ronald Haring), Шило Морриса (Shiloh Morri), Винсента Терона (Vincent Theron) и Уильяма Э. Уилера (William E. Wheeler).

Хочу также поблагодарить сотрудников издательства Manning: Дейрдре Хиам (Deirdre Hiam), Фрэнсиса Бурана (Frances Buran), Кери Хейлза (Keri Hales), Дэвида Новака (David Novak), Мелоди Долаб (Melody Dolab) и Николь Берд (Nichole Beard).

О книге

Кому адресована эта книга

Цель этой книги – не просто помочь выучить язык Kotlin, но также научить писать гораздо более безопасные программы. Это не означает, что Kotlin следует использовать, только если вы решите писать более безопасные программы, и тем более не означает, что писать более безопасные программы можно только на Kotlin. Эта книга представляет примеры, написанные на Kotlin, потому что это один из самых дружественных языков для разработки безопасных программ в экосистеме JVM.

В книге рассказывается о методах, разработанных давно и в самых разных окружениях, хотя многие из них своими корнями уходят в функциональное программирование. Но эта книга не о фундаменталистском функциональном программировании, она о прагматичном безопасном программировании.

Все описанные здесь приемы годами использовались в экосистеме Java и доказали свою эффективность в разработке программ с гораздо меньшим количеством ошибок реализации, чем традиционные методы императивного программирования. Эти безопасные приемы можно реализовать на любом языке, и некоторые из них уже многие годы используются в Java, но часто для этого приходится преодолевать ограничения Java.

Эта книга не учит программированию с самого начала. Она писалась для профессиональных программистов, ищущих более простые и безопасные способы разработки безошибочных программ.

О чем вы узнаете

В этой книге вы познакомитесь с конкретными приемами, которые могут отличаться от уже знакомых вам, если прежде вы программировали на Java. Большинство из них покажутся вам незнакомыми или даже противоречащими приемам, которые программисты привыкли считать

оптимальными. Но многие (хотя и не все) оптимальные приемы были выработаны во времена, когда компьютеры имели 640 Кб памяти, 5 Мб дискового пространства и одноядерный процессор. Времена изменились. Сейчас простой смартфон – это настоящий компьютер с 3 Гб или более оперативной памяти, с твердотельным накопителем на 256 Гб и 8-ядерным процессором. Компьютеры тоже имеют много гигабайт оперативной памяти, терабайты дискового пространства и многоядерные процессоры.

В этой книге я расскажу о:

- дальнейшем увеличении абстракции;
- предпочтительном использовании неизменяемых объектов;
- ссылочной прозрачности;
- инкапсуляции общего изменяемого состояния;
- абстрагировании условных и управляющих структур;
- использовании правильных типов данных;
- использовании отложенных вычислений;
- и многом другом.

Дальнейшее увеличение абстракции

Один из наиболее важных методов, с которыми вы познакомитесь, – это дальнейшее увеличение абстракции (традиционные программисты привыкли считать преждевременную абстракцию злом, как и преждевременную оптимизацию). Но дальнейшее увеличение абстракции помогает лучше понять решаемую задачу, что в свою очередь гарантирует правильность решения.

Вы можете спросить, что в действительности подразумевается под дальнейшим увеличением абстракции. Здесь все просто – это умение распознавать типичные шаблоны в различных вычислениях и их абстрагирование, чтобы избежать повторяющегося кода.

Неизменяемость

Суть неизменяемости заключается в использовании только неизменяемых данных. Многим традиционным программистам трудно представить, как можно писать полезные программы, используя только неизменные данные. Разве программирование не основано в первую очередь на изменении данных? Если следовать такой логике, тогда бухгалтерский учет – это прежде всего изменение значений в бухгалтерской книге.

Переход от использования изменяемого к использованию неизменяемого учета произошел еще в XV веке, и с тех пор принцип неизменяемости был признан основным элементом безопасности для учета. Этот принцип также применим к программированию, как вы увидите в этой книге.

Ссылочная прозрачность

Ссылочная прозрачность позволяет писать детерминированные программы, т. е. программы, результаты работы которых можно предсказать

и осмыслить. Такие программы всегда дают одинаковые результаты для одних и тех же входных данных. Это не означает, что они всегда дают одинаковые результаты, но различия в результатах зависят только от изменений на входе, а не от внешних условий.

Такие программы не только безопаснее (потому что обладают предсказуемым поведением), но их также проще писать, сопровождать, обновлять и тестировать. А программы, которые легче тестировать, обычно проверяются более полно и, следовательно, получаются более надежными.

Инкапсуляции общего изменяемого состояния

Неизменяемые данные автоматически оказываются защищены от случайного изменения при совместном использовании, что часто вызывает много проблем при конкурентной и параллельной обработке данных, таких как взаимоблокировка, простои потоков выполнения и устаревание данных. Но отсутствие общего изменяемого состояния превращается в проблему, когда оно действительно необходимо. Это в первую очередь относится к конкурентному и параллельному программированию.

Устранением изменяемого состояния исключается возможность случайного обмена ошибочными данными, благодаря чему программы становятся безопаснее. Но конкурентное и параллельное программирование подразумевает совместное изменение общего состояния. Без этого конкурирующие и параллельные потоки выполнения не смогут сотрудничать друг с другом. Этот конкретный вариант использования изменяемого общего состояния можно абстрагировать и инкапсулировать так, чтобы получить возможность повторного использования без риска, потому что будет иметься одна полностью протестированная универсальная реализация.

В этой книге вы узнаете, как абстрагировать и инкапсулировать общее изменяемое состояние, чтобы реализовать его только один раз и потом использовать везде, где оно необходимо.

Абстрагирование условных и управляющих структур

Вторым распространенным источником ошибок в программах после общего изменяемого состояния являются управляющие структуры. Традиционные программы состоят из таких управляющих структур, как циклы и проверки условий. С этими структурами так легко ошибиться, что разработчики языка постарались максимально абстрагировать детали. Одним из лучших примеров является цикл `for-each`, который ныне присутствует в большинстве языков (хотя в Java он все еще называется `for`).

Другая распространенная проблема – правильное использование `while` и `do while` (или `repeat until`) и, в частности, выбор места, где должно проверяться условие. Еще одна проблема – конкурентное изменение элементов коллекций во время их обхода в цикле, когда проблема общего изменяемого состояния возникает даже при использовании единственного потока выполнения! Абстрагирование управляющих структур позволяет полностью устранить подобные проблемы.

Использование правильных типов данных

В традиционном программировании такие универсальные типы, как `int` и `String`, используются для представления величин без учета единиц измерения. Как следствие, очень легко допустить ошибку, сложив мили с галлонами или доллары с минутами. Использование типов значений может полностью устранить проблемы такого рода при очень низких затратах, даже если используемый язык не предлагает истинных типов значений.

Отложенные вычисления

Большинство распространенных языков называют *строгими*, в том смысле, что аргументы, передаваемые методу или функции, вычисляются заранее, перед их обработкой. На первый взгляд, такое поведение вполне оправданно, хотя часто это не так. Отложенные, или «ленивые», вычисления – это прием, заключающийся в откладывании вычисления элементов до момента, когда они действительно становятся необходимы. Программирование почти целиком основано на отложенных вычислениях.

Например, условие в конструкции `if...else` вычисляется немедленно, перед его проверкой, но выполнение ветвей откладывается, в том смысле, что выполняется только ветвь, соответствующая условию. Эта отложенность скрыта, и программист не контролирует ее. Явное использование отложенных вычислений помогает писать гораздо более эффективные программы.

Читатели

Эта книга адресована читателям, уже имеющим опыт программирования на Java. Необходимо также некоторое понимание параметризованных (обобщенных) типов. В этой книге широко используются параметризованные функции, или варианты, которые редко применяются в Java (хотя это мощный инструмент). Не пугайтесь, если вы еще не знакомы с этими приемами; я расскажу о них и объясню, зачем они нужны, когда подойдет время.

Структура книги

Книга предполагает последовательное чтение, потому что каждая следующая глава основана на понятиях, рассматриваемых в предыдущих главах. Я использую слово «чтение», но эта книга предназначена не только для чтения. Очень немногие разделы содержат одну лишь теорию.

Чтобы извлечь максимум пользы из книги, выполняйте все упражнения, встречающиеся в процессе чтения. Каждая глава содержит ряд упражнений с необходимыми инструкциями и подсказками, которые помогут вам прийти к решению. Каждое упражнение сопровождается предлагаемым решением и тестом, который вы можете использовать для проверки своего решения.

ПРИМЕЧАНИЕ Программный код примеров доступен бесплатно на GitHub (http://github.com/pysaumont/fpinkotlin). В него включены все элементы, необходимые для импортирования проекта в IntelliJ (рекомендуется) или для компиляции и выполнения с использованием Gradle 4. Пользующиеся Gradle могут редактировать код в любом текстовом редакторе. Предполагается, что Kotlin можно использовать в Eclipse, но я не могу этого гарантировать. IntelliJ – намного более удобная интегрированная среда разработки (IDE), которую можно бесплатно получить на сайте Jetbrains (https://www.jetbrains.com/idea/download).

Выполнение упражнений

Упражнения играют важную роль в обучении и помогают понять то, о чем рассказывает эта книга. В процессе чтения теоретических разделов какие-то идеи и понятия могут остаться для вас непонятными, и в этом нет ничего страшного. Но выполнение упражнений играет особенно важную роль в процессе обучения, поэтому я призываю вас не пропускать никаких упражнений.

Некоторые могут показаться довольно сложными, и у вас может возникнуть соблазн заглянуть в предложенные решения. Это нормально, но после этого вернитесь к упражнению и выполните его, не заглядывая в решение. Если вы ограничитесь простым просмотром решения, у вас почти наверняка возникнут проблемы при попытке выполнить более сложные упражнения в последующих главах.

Вам не придется вводить много кода, потому что большинство упражнений состоит в написании реализаций функций, для которых вы получаете готовые окружение и сигнатуру функции. Ни одно упражнение не потребует ввода больше десятка строк кода; а в большинстве случаев решения состоят из четырех или пяти строк. Закончив упражнение (т. е. добившись безошибочной компиляции вашей реализации), просто запустите соответствующий тест, чтобы убедиться в правильности решения.

Важно отметить, что каждое упражнение является самодостаточным и независящим от остальной части главы, поэтому код, представленный внутри главы, повторяется от упражнения к упражнению. Это необходимо, потому что часто следующее упражнение основано на предыдущем. По этой причине реализации могут отличаться даже при использовании одного и того же класса. Как следствие, старайтесь не заглядывать в последующие упражнения, не завершив предыдущие, потому что вы увидите решения для еще не решенных упражнений.

Обучение методикам в этой книге

Методики, описанные в этой книге, не сложнее традиционных. Они просто разные. Те же задачи можно решать с использованием традиционных методик, но перевод решения с одной методики на другую иногда может сопровождаться потерей эффективности.

Изучение новых методик сродни изучению иностранного языка. Как нельзя эффективно мыслить на одном языке и переводить на другой, так же нельзя рассуждать терминами традиционного программирования, основанного на изменении состояния и потоке управления, и преобразовывать свой код в функции, обрабатывающие неизменяемые данные. И точно так же, как вы должны научиться думать на новом языке, вы должны научиться думать по-другому. Этого нельзя добиться простым чтением; это приходит с написанием кода. Поэтому обязательно практикуйтесь!

Вот почему я не ожидаю, что вы придете к полному пониманию написанного в этой книге, просто прочитав ее, и даю так много упражнений; *старайтесь* выполнять все упражнения, чтобы полностью понять представленные идеи. Сами по себе темы несложные, но, чтобы понять их, одного только чтения недостаточно. Если бы вы могли понять суть без выполнения упражнений, тогда вам, вероятно, не понадобилась бы эта книга.

Решение упражнений – это ключ к успешному освоению методик, описываемых в этой книге. Я советую стараться решать все встречающиеся упражнения по порядку, ничего не пропуская, прежде чем продолжать чтение. Если у вас не получится найти решение, попробуйте еще раз, и, только если у вас ничего не получится со второй попытки, переходите к предлагаемым решениям.

Если вам сложно понять что-то, задайте вопрос на форуме (ссылка приводится в следующем разделе). Задавая вопросы и получая ответы на форуме, вы не только помогаете себе, но также человеку, отвечающему на вопрос (и многим другим, у кого может возникнуть та же проблема). Мы в большей мере учимся, когда отвечаем на вопросы (в основном на свои собственные), а не когда задаем их.

Исходный программный код

Книга содержит много примеров исходного кода в виде листингов и фрагментов в обычном тексте. В обоих случаях исходный код оформляется моноширинным шрифтом, чтобы его можно было отличить от обычного текста.

Во многих случаях оригинальный исходный код был переформатирован; мы добавили переносы строк и изменили ширину отступов, чтобы уместить строки кода по ширине книжной страницы. Многие листинги сопровождаются дополнительными аннотациями, подчеркивающими наиболее важные идеи.

Исходный код можно загрузить в виде файла архива или клонировать с помощью Git. Код для упражнений организован в модули с именами, отражающими названия глав, а не их номера. По этой причине IntelliJ отсортирует их в алфавитном порядке, а не в порядке следования в книге.

Чтобы помочь вам понять, какой модуль к какой главе относится, я подготовил список глав с соответствующими именами модулей в фай-

ле *README* (http://github.com/pysaumont/fpinkotlin).

Исходный код из всех листингов в книге доступен также для загрузки на веб-сайте издательства Manning: https://www.manning.com/books/the-joy-ofkotlin.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста,

сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmk-press@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторе

Пьер-Ив Сомон (Pierre-Yves Saumont) – опытный разработчик на Java с тридцатилетней практикой проектирования и создания корпоративного программного обеспечения. Работает инженером-исследователем в ASN (Alcatel Submarine Networks).

Об иллюстрации на обложке

На обложке «Волшебства Kotlin» изображен рисунок, озаглавленный как «Традиционный женский костюм в Тартарии, 1700». Рисунок взят из книги «Collection of the Dresses of Different Nations, Ancient and Modern» (Коллекция костюмов разных народов, античных и современных) Томаса Джефферис (Thomas Jefferys), опубликованной в Лондоне между 1757 и 1772 годами. На титульной странице указано, что это выполненная вручную каллиграфическая цветная гравюра, обработанная гуммиарабиком.

Томас Джефферис (1719–1771) носил звание «географа короля Георга III». Английский картограф, он был ведущим поставщиком карт того времени. Он выгравировал и напечатал множество карт для нужд правительства и других официальных органов, широкий спектр коммерческих карт и атласов, в частности, Северной Америки. Будучи картографом, интересовался местной одеждой народов, населяющих разные земли, и собрал блестящую коллекцию различных платьев в четырех томах. Очарование далекими землями и дальние путешествия для удовольствия были относительно новым явлением в конце XVIII века, и коллекции, такие как эта, были весьма популярны, так как знакомили с внешним видом жителей других стран.

Разнообразие рисунков, собранных Джефферисом, свидетельствует о проявлении народами мира около 200 лет яркой индивидуальности и уникальности. С тех пор стиль одежды сильно изменился и исчезло разнообразие, характеризующее различные области и страны. Теперь трудно отличить по одежде даже жителей разных континентов. Если взглянуть на это с оптимистической точки зрения, мы пожертвовали культурным и внешним разнообразием в угоду разнообразию личной жизни или в угоду более разнообразной и интересной интеллектуальной и технической деятельности.

В наше время, когда трудно отличить одну техническую книгу от другой, издательство Manning проявило инициативу и деловую сметку, украшая обложки книг изображениями, основанными на богатом разнообразии жизненного уклада народов двухвековой давности, придав новую жизнь рисункам Джефферис.

1 Создание безопасных программ

Эта глава охватывает следующие темы:

- выявление программных ловушек;
- проблемы с эффектами;
- как ссылочная прозрачность увеличивает безопасность программ;
- использование подстановочной модели в рассуждениях о программе;
- максимальное использование абстракций.

Программирование – опасная работа. Если вы программист-любитель, вас наверняка удивит такое утверждение. Вы, вероятно, думали, что находитесь в безопасности, сидя перед экраном и клавиатурой. Вы можете думать, что рискуете не более чем заработать боль в спине, если сидеть слишком долго, некоторые проблемы со зрением при чтении мелких символов на экране или даже тендинит запястья, если вам случается слишком много печатать. Но если вы профессиональный программист (или хотите им стать), реальность намного хуже.

Главная опасность – ошибки, скрывающиеся в ваших программах. Ошибки могут стоить дорого, если появятся не вовремя. Помните ошибку Y2K? Во многих программах, написанных между 1960 и 1990 годами, для обозначения года в датах использовались только две цифры, поскольку программисты не ожидали, что их программы доработают до следующего столетия. Многие из этих программ, все еще использовавшихся

в 1990-х годах, рассматривали бы 2000 год как 1900. Ориентировочная стоимость этой ошибки в долларах США в 2017 году составила 417 млрд[1].

Но иногда ошибка, возникающая в одной программе, может стоить намного выше. 4 июня 1996 года первый полет французской ракеты Ariane 5 закончился аварией через 36 с после старта. Сбой произошел из-за ошибки в системе навигации. Одно целочисленное арифметическое переполнение привело к потере 370 млн долл[2].

Как бы вы себя чувствовали, если бы были привлечены к ответственности за такую катастрофу? Как бы вы себя чувствовали, если бы постоянно писали такие программы и не были уверены, что программа, работающая сегодня, будет работать и завтра? Именно так работает большинство программистов: они пишут недетерминированные программы, которые возвращают разные результаты при запуске с одними и теми же входными данными. Пользователи знают об этом, и когда программа возвращает результаты, отличающиеся от ожидаемых, они повторяют попытку, надеясь, что недетерминированность приведет к другому результату. И иногда это случается, потому что никто не знает, от чего зависит результат, возвращаемый программой.

С развитием искусственного интеллекта (ИИ) проблема надежности программного обеспечения стала еще более актуальной. Если программы, принимающие решения, такие как автопилоты в самолетах или в беспилотных автомобилях, могут поставить под угрозу человеческую жизнь, мы должны быть уверены, что они работают в точности, как задумано.

Что нужно, чтобы сделать программы более безопасными? Некоторые ответят, что нужны лучшие программисты. Но хорошие программисты подобны хорошим водителям. 90 % программистов согласны с тем, что только 10 % достаточно хороши, но при этом 90 % программистов считают, что они являются частью 10 %!

Самое необходимое качество для программистов – умение признавать собственные ограничения. Посмотрим правде в глаза: мы – всего лишь средние программисты. Мы тратим 20 % времени на написание программ с ошибками, а затем 40 % времени – на рефакторинг кода, чтобы получить программы без видимых ошибок. А позже мы тратим еще 40 % на отладку кода, который уже находится в эксплуатации, потому что ошибки делятся на две категории: очевидные и неочевидные. Вне всяких сомнений, неочевидные ошибки станут очевидными – это всего лишь вопрос времени. Остается вопрос: как долго и насколько большой будет наноситься ущерб, прежде чем ошибки станут очевидными.

Как можно решить эту проблему? Никакой инструмент, методика или дисциплина не могут гарантировать отсутствие ошибок в наших программах. Но есть методики, которые могут устранить отдельные катего-

---

[1] Согласно оценке потребительских цен федерального резервного банка: «Consumer Price Index (estimate) 1800–» (https://www.minneapolisfed.org/community/teaching-aids/cpi-calculator-information/consumer-price-index-1800).

[2] Отчет комиссии по расследованию аварии Ariane 501 (http://www.astrosurf.com/luxorion/astronautique-accident-ariane-v501.htm).

рии ошибок и гарантировать, что оставшиеся ошибки будут присутствовать только в ограниченных (небезопасных) областях наших программ. Это имеет огромное значение, потому что делает поиск ошибок намного проще и эффективнее. Среди таких методик находится рекомендация писать простые программы, в которых отсутствие ошибок очевидно, вместо сложных программ, где не видно очевидных ошибок[1].

В оставшейся части главы я кратко познакомлю вас с такими понятиями, как неизменяемость, ссылочная прозрачность и подстановочная модель, а также дам несколько рекомендаций, которые помогут вам сделать свои программы намного безопаснее. В следующих главах мы будем применять эти идеи снова и снова.

1.1 Программные ловушки

Программирование часто рассматривают как способ описания выполнения некоторого процесса. Такое описание обычно включает перечисление действий, изменяющих состояние программной модели с целью выполнения поставленной задачи и принятия решения о результатах таких изменений. Так видит программирование большинство, даже если это не программисты.

Если перед вами поставили слишком сложную задачу, вы делите ее на шаги. Затем выполняете первый шаг, проверяете результат и по результатам проверки выбираете следующий шаг для выполнения. Например, программу сложения двух положительных значений `a` и `b` можно представить следующим псевдокодом:

- если b = 0, вернуть a;
- иначе увеличить a и уменьшить b;
- начать с начала с новыми значениями a и b.

В этом псевдокоде можно заметить инструкции, традиционные для большинства языков программирования: проверка условия, изменение переменной, ветвление и возврат значения. Этот код можно представить графически, в виде блок-схемы, как показано на рис. 1.1.

Легко заметить, где в этой программе могут возникнуть проблемы. Измените любые данные на блок-схеме или начальный или конечный пункт любой стрелки, и вы получите программу с ошибкой. Если повезет, вы получите программу, которая вообще не запускается или работает вечно и никогда не останавливается. Это можно считать удачей, потому что вы сразу увидите, что в программе есть проблема, которую нужно исправить. На рис. 1.2 показаны три такие проблемы.

---

[1] «…есть два способа конструирования программного обеспечения: один из них заключается в том, чтобы сделать его настолько простым, чтобы в нем явно не было недостатков, а другой – сделать его настолько сложным, чтобы в нем не было явных недостатков. Первый способ гораздо сложнее». *С.А.Р. Хоар* (C.A.R. Hoare). The Emperor's Old Clothes // Communications of the ACM 24 (февраль 1981): с. 75–83.

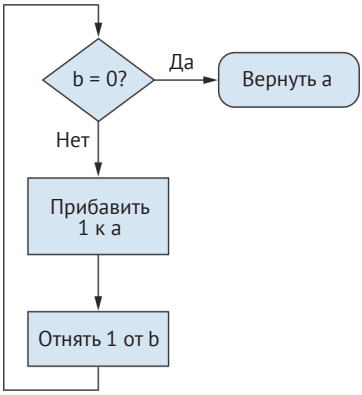


Рис. 1.1 Блок-схема, представляющая программу как процесс, протекающий во времени. В ходе выполнения программы выполняются разные преобразования и изменение состояния, пока наконец не будет получен результат

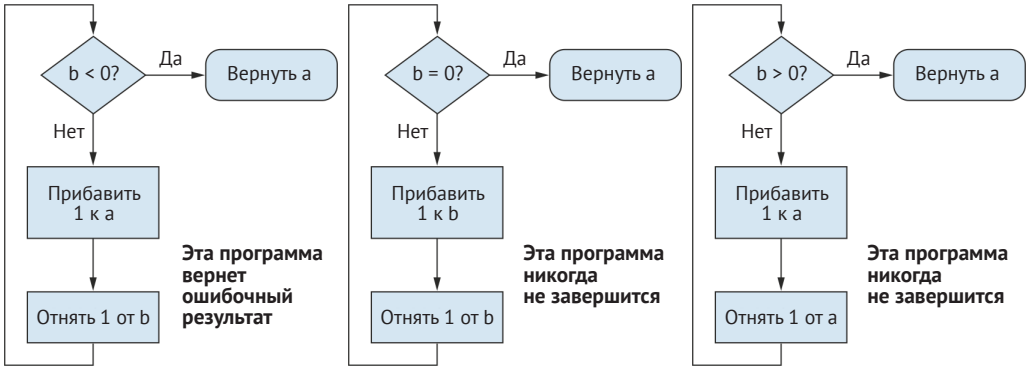


Рис. 1.2 Три версии программы с ошибками

Первая версия возвращает неверный результат, а вторая и третья никогда не завершатся. Однако обратите внимание, что отдельные языки программирования могут не позволить написать некоторые из этих версий. Ни одну из них нельзя написать на языке, который не допускает изменения значения по ссылке, и ни одну из них нельзя написать на языке, который не поддерживает ветвление или циклы. Вы можете подумать, что для устранения проблемы достаточно лишь использовать такой язык. Конечно, можно пойти этим путем. Но вы будете ограничены небольшим количеством языков, и может так получиться, что ни один из них не будет разрешен для использования в вашей профессиональной сфере.

Есть другое решение? Да, есть: отказаться от использования ссылок, допускающих возможность изменения, ветвления (если ваш язык это позволяет) и циклов. То есть нужно просто выработать определенную дисциплину программирования.

Не используйте потенциально опасные возможности, такие как изменение значений переменных и циклы. Это так просто! А если обнаружится, что без изменяемых ссылок или циклов не обойтись, абстрагируйте их. Напишите какой-нибудь компонент, который раз и навсегда скроет

в своих недрах изменение состояния, и вы навсегда избавитесь от проблемы. (Некоторые более или менее экзотические языки предлагают такие компоненты «из коробки», но, вероятно, это тоже не те языки, которые вы сможете использовать в своей сфере.) То же относится к циклам. Большинство современных языков наряду с традиционными циклами предлагают абстракции циклов. И снова использовать или не использовать циклы – это вопрос дисциплины. Применяйте только качественные детали! Подробнее об этом в главах 4 и 5.

Другой распространенный источник ошибок – пустые (`null`) ссылки. Как будет показано в главе 6, Kotlin позволяет ясно отделить код, который допускает пустые ссылки, от кода, который запрещает их. Но вообще желательно полностью искоренить использование пустых ссылок в программах.

Многие ошибки вызваны зависимостями программ от внешнего мира. Но зависимости от внешнего мира необходимы почти во всех программах. Ограничение влияния этих зависимостей конкретными областями в ваших программах упростит поиск и устранение проблем, хотя и не избавит полностью от появления ошибок такого типа.

В этой книге вы познакомитесь с несколькими правилами, помогающими сделать программы намного безопаснее. Вот их список:

- не используйте изменяемые ссылки (переменные) и абстрагируйте единичные случаи, когда отказаться от изменения состояния не получается;
- не используйте управляющие конструкции;
- ограничьте эффекты (взаимодействия с внешним миром) определенными областями в коде. То есть не нужно выводить что-либо в консоль или на любое другое устройство, записывать данные в файлы, базы данных, сети или куда-либо еще за пределами этих ограниченных областей;
- не возбуждайте исключений. Возбуждение исключения – это современная форма ветвления (`GOTO`), из-за которого код начинает напоминать *тарелку со спагетти,* – трудно понять, где и что начинается, и невозможно проследить, куда движется выполнение. В главе 7 вы узнаете, как избежать возбуждения исключений.

1.1.1 Безопасная обработка эффектов

Как я уже отмечал, под «эффектами» подразумеваются взаимодействия с внешним миром, такие как вывод в консоль, запись в файл, в базу данных или в сеть, а также изменение чего-то, находящегося вне области видимости компонента. Программы обычно пишутся небольшими блоками, имеющими свои области видимости. В некоторых языках эти блоки называются *процедурами*; в других (например, Java) – *методами*. В Kotlin они называются *функциями*, хотя это название несет несколько иной смысл, чем понятие функции в математике.

Функции в Kotlin фактически являются методами, как в Java и многих других современных языках. Эти блоки кода имеют *область видимости*,

т. е. область программы, видимую только этими блоками. Блоки имеют не только свою, огражденную от других область видимости, но и доступ к внешним областям видимости и – посредством их – к внешнему миру. Следовательно, любое изменение внешнего мира, вызванное функцией или методом (например, изменение в области видимости класса, в котором определен метод), является эффектом.

Некоторые методы (функции) возвращают значение. Одни изменяют внешний мир, а другие делают и то, и другое. Когда метод или функция возвращает значение и вызывает эффект, такой эффект называется *побочным*. Появление побочных эффектов в программировании всегда считалось неправильным. В медицине термин «побочные эффекты» часто используется для описания нежелательных результатов лечения. В программировании побочный эффект – это явление, наблюдаемое за пределами программы, *дополняющее* результат, возвращаемый программой.

Если программа не возвращает результата, нельзя сказать, что наблюдаемый эффект является побочным; это главный эффект. Однако он может сопровождаться побочными (вторичными) эффектами, которые также часто считаются нежелательными с точки зрения принципа «единственной ответственности».

Безопасные программы конструируются из функций, которые принимают аргумент и возвращают значение, и все. Нас не волнует, что происходит *внутри* функций, потому что теоретически там никогда ничего не происходит. Некоторые языки предлагают только такие функции без эффектов: программы, написанные на этих языках, не имеют видимых эффектов, кроме возвращаемого значения. Но в действительности это значение может быть новой программой, которую можно запустить и получить эффект. Такой подход можно использовать в любом языке, но часто он считается неэффективным (что спорно). Безопасной альтернативой является ясное отделение кода, производящего эффект, от остальной части программы и даже, насколько это возможно, его абстрагирование. С несколькими приемами, позволяющими сделать это, вы познакомитесь в главах 7, 11 и 12.

1.1.2 Увеличение безопасности программ за счет ссылочной прозрачности

Отсутствия побочных эффектов (изменений во внешнем мире) недостаточно, чтобы сделать программу безопасной и детерминированной. Программы также не должны подвергаться влиянию внешнего мира – результат их работы должен зависеть только от аргументов. То есть программы не должны читать данные из консоли, файла, сети, базы данных или даже из переменных системы.

Код, который ничего не изменяет и не зависит от внешнего мира, называется *ссылочно-прозрачным* (referentially transparent). Ссылочно-прозрачный код обладает рядом интересных свойств:

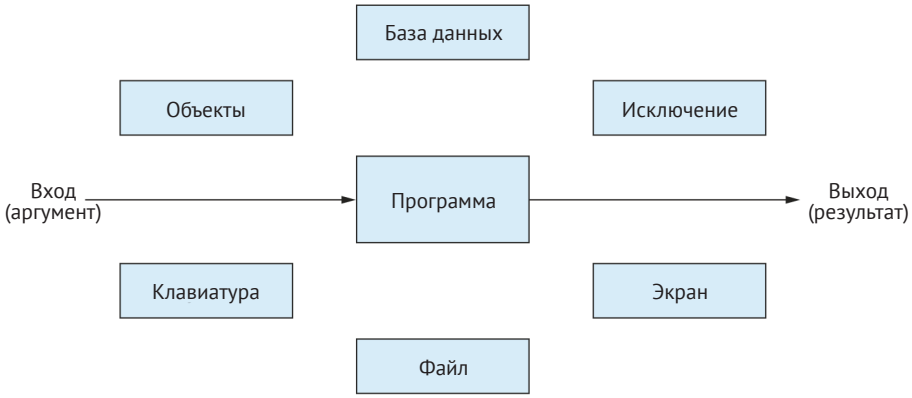
- *самодостаточность* – его можно использовать в любом контексте, достаточно лишь передать ему допустимые аргументы;
- *детерминированность (определенность)* – для одних тех же аргументов он всегда возвращает одно и то же значение; он может возвращать ошибочный результат, но, по крайней мере, результат всегда будет одинаковым для одного и того же аргумента;
- *никогда не возбуждает исключений* – он может вызывать ошибки, такие как нехватка памяти или переполнение стека, но эти ошибки явно указывают на ошибки в коде. Это не та ситуация, которую вы, как программист, или пользователи вашего API должны обрабатывать (но важно предусмотреть защиту от аварийного завершения приложения, которая часто не дается автоматически, и в конечном итоге исправить ошибку);
- *не создает условий, вызывающих неожиданный сбой другого кода,* – например, он не изменяет аргументы и любые другие внешние данные, из-за чего вызывающая сторона могла бы столкнуться с проблемой неактуальных данных или с исключениями одновременного доступа;
- *не зависит от какого-либо внешнего устройства* – он не зависнет из-за того, что какое-то внешнее устройство (база данных, файловая система или сеть) недоступно, работает слишком медленно или вышло из строя.

На рис. 1.3 показаны различия между ссылочно-прозрачными и непрозрачными программами.

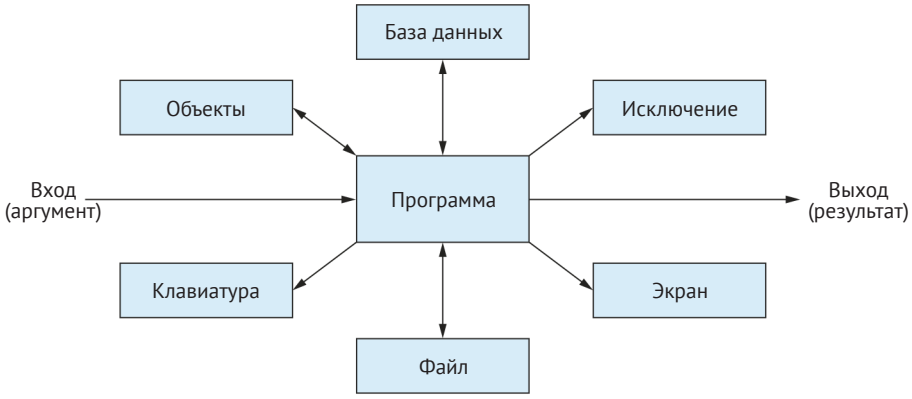
1.2 Выгоды безопасного программирования

По моему описанию выше вы наверняка догадаетесь, какие выгоды несет ссылочная прозрачность:

1. *Ссылочно-прозрачные программы проще анализировать благодаря их детерминированности*. Конкретные входные аргументы всегда будут приводить к одному и тому же результату. Во многих случаях правильность таких программ можно доказать, не проводя тщательного тестирования и не опасаясь, что она выполнится как-то не так при необычных условиях.
2. *Ссылочно-прозрачные программы проще тестировать*. Поскольку такие программы не имеют побочных эффектов, вам не понадобятся имитации, которые часто требуются при тестировании для изоляции компонентов программы от внешнего мира.
3. *Ссылочно-прозрачные программы имеют более модульную организацию*. Причина в том, что такие программы строятся из функций, имеющих только вход и выход, – нет никаких побочных эффектов, никаких исключений и изменений контекста, которые нужно обрабатывать, нет общего изменяемого состояния и никаких операций, одновременно изменяющих одно и то же состояние.



Ссылочно-прозрачная программа не влияет на внешний мир – она лишь принимает аргумент на входе и возвращает результат на выходе. Результат зависит только от аргумента



Программа, не являющаяся ссылочно-прозрачной, может читать данные из внешнего мира или записывать их во внешние элементы или в файлы, изменять внешние объекты, читать с клавиатуры, выводить на экран и т. д. Результат работы такой программы непредсказуем

Рис. 1.3 Сравнение ссылочно-прозрачной и ссылочно-непрозрачной программ

4 *Составлять и реорганизовывать ссылочно-прозрачные программы намного проще*. Создание такой программы начинается с разработки различных базовых функций, а затем эти функции объединяются в функции более высокого уровня. Этот процесс повторяется, пока не будет получена единственная функция, представляющая всю программу. А так как все эти функции являются ссылочно-прозрачными, их можно использовать для создания других программ без каких-либо изменений.

5 *Ссылочно-прозрачные программы по своей природе поддерживают многопоточное выполнение, потому что не изменяют общего состояния*. Это не означает, что все данные должны быть неизменными, – неизменными должны быть только общие данные. Программисты, применяющие это правило, вскоре понимают, что неизменяемые

данные всегда безопаснее, даже если изменения не видны снаружи. Одна из причин состоит в том, что данные, не являющиеся общими, не станут случайно общедоступными после рефакторинга. Повсеместное использование неизменяемых данных гарантирует, что такого рода проблемы никогда не возникнут.

В оставшейся части этой главы я представлю несколько примеров, как использование ссылочной прозрачности помогает писать безопасные программы.

1.2.1 Использование подстановочной модели в рассуждениях о программе

Основным преимуществом использования функций, которые возвращают значение без любого другого наблюдаемого эффекта, является их эквивалентность возвращаемому значению. Такая функция ничего не делает. Она имеет значение, зависящее только от ее аргументов. Как следствие, вызов функции или любое ссылочно-прозрачное выражение всегда можно заменить его значением, как показано на рис. 1.4.

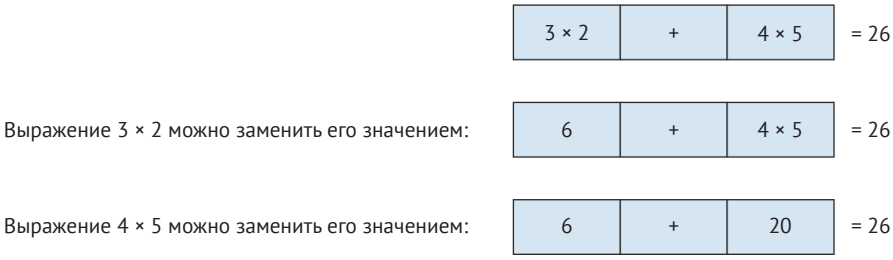


Рис. 1.4 Замена ссылочно-прозрачных выражений их значениями не меняет сути дела

Применительно к функциям прием подстановочного моделирования позволяет заменить вызов любой функции ее возвращаемым значением. Взгляните на следующий код:

```
fun main(args: Array<String>) {
    val x = add(mult(2, 3), mult(4, 5))
    println(x)
}

fun add(a: Int, b: Int): Int {
    log(String.format("Returning ${a + b} as the result of $a + $b"))
    return a + b
}

fun mult(a: Int, b: Int) = a * b

fun log(m: String) {
    println(m)
}
```

Замена вызовов `mult(2, 3)` и `mult(4, 5)` соответствующими возвращаемыми значениями не влияет на результат работы программы, которая упрощается до выражения

```
val x = add(6, 20)
```

Замена вызова функции `add` ее возвращаемым значением, напротив, существенно влияет на результат программы, потому что в этом случае функция `log` не будет вызвана и регистрация результата не произойдет. Это, может быть, и не важно, но, как бы то ни было, такая подстановка меняет результат программы.

1.2.2 Применение принципов соблюдения безопасности на простом примере

Чтобы увидеть, как преобразовать небезопасную программу в более безопасную, рассмотрим простой пример реализации покупки пончика с помощью кредитной карты.

Листинг 1.1. Программа на Kotlin с побочными эффектами

```
fun buyDonut(creditCard: CreditCard): Donut {
  val donut = Donut()
  creditCard.charge(donut.price) ①
  return donut ②
}
```

① **Взимает плату с кредитной карты – это побочный эффект**
② **Возвращает пончик**

В этом коде взимание платы с кредитной карты является побочным эффектом. Взимание платы, вероятно, включает обращение в банк, проверку действительности кредитной карты, авторизацию и регистрации транзакции. Функция возвращает пончик.

Проблема с подобным кодом в том, что его сложно тестировать. Тестовый прогон программы должен включать обращение в банк и регистрацию транзакции с использованием некоторого фиктивного счета. Или вам придется создать фиктивную кредитную карту, чтобы зарегистрировать эффект вызова функции `charge` и проверить ее состояние после тестирования.

Чтобы иметь возможность проверить программу без обращения в банк или без создания фиктивной карты, необходимо устранить побочный эффект. Но так как программа все же должна снимать средства с кредитной карты, единственное решение – добавить представление этой операции в возвращаемое значение. В данном случае функция `buyDonut` должна возвращать не только пончик, но и представление платежа. Для представления платежа можно использовать класс `Payment`, как показано в листинге 1.2.

Листинг 1.2. Класс Payment

```
class Payment(val creditCard: CreditCard, val amount: Int)
```

Этот класс содержит все данные, необходимые для представления платежа, включая сведения о кредитной карте и сумму для списания. Поскольку функция `buyDonut` должна возвращать `Donut` и `Payment`, с этой целью можно создать отдельный класс, например `Purchase`.

```
class Purchase(val donut: Donut, val payment: Payment)
```

Вам часто будут нужны такие классы, предназначенные для хранения двух (или более) значений разного типа, потому что, чтобы сделать программы безопаснее, нужно заменить побочные эффекты возвратом представления этих эффектов.

Вместо создания специализированного класса `Purchase` можно использовать обобщенный класс `Pair`. Этот класс параметризуется двумя типами, в данном случае `Donut` и `Payment`. Кроме `Pair`, Kotlin предлагает также класс `Triple`, позволяющий представить три значения. Такой класс мог бы очень пригодиться в языке, подобном Java, потому что определение класса `Purchase` в Java подразумевает объявление конструктора, методов чтения и, возможно, методов `equals` и `hashCode`, а также `toString`. Но его ценность в Kotlin несколько ниже, потому что тот же результат можно получить всего одной строкой кода:

```
data class Purchase(val donut: Donut, val payment: Payment)
```

Класс `Purchase` уже не нуждается в явном конструкторе и методах чтения. А если добавить ключевое слово `data` перед определением класса, Kotlin автоматически включит в класс реализации по умолчанию `equals`, `hashCode`, `toString` и `copy`. Два экземпляра класса данных будут считаться равными, только если все их свойства будут равны. Если понятие «равно» в вашем случае имеет другой смысл, вам придется переопределить эти функции своими реализациями.

```
fun buyDonut(creditCard: CreditCard): Purchase {
    val donut = Donut()
    val payment = Payment(creditCard, Donut.price)
    return Purchase(donut, payment)
}
```

На данном этапе вас больше не волнует проблема взимания платы с кредитной карты. Это добавляет некоторую свободу выбора при создании приложения. Можно обработать платеж немедленно или сохранить его для дальнейшей обработки. Можно даже объединить сохраненные платежи с одной и той же карты и обработать их в одной операции. Это поможет сэкономить деньги за счет уменьшения банковских сборов за обслуживание кредитной карты.

В листинге 1.3 показана функция `combine`, объединяющая платежи. Если кредитные карты не совпадают, она возбуждает исключение. Это не противоречит тому, что я говорил о безопасных программах, не генерирующих исключений. Здесь попытка объединить два платежа с двумя разными кредитными картами считается ошибкой, поэтому она должна вызывать сбой приложения. (Впрочем, это не самый лучший выход. Но подождите до главы 7, где я расскажу, как справляться с такими ситуациями без создания исключений.)

Листинг 1.3. Объединение нескольких платежей

```
package com.fpinkotlin.introduction.listing03
class Payment(val creditCard: CreditCard, val amount: Int) {
    fun combine(payment: Payment): Payment =
        if (creditCard == payment.creditCard)
            Payment(creditCard, amount + payment.amount)
        else
            throw IllegalStateException("Cards don't match.")
}
```

В этом сценарии функция `combine` будет не самым эффективным решением при покупке сразу нескольких пончиков. Более оптимальное решение: заменить функцию `buyDonut` функцией `buyDonuts(n: Int, creditCard: CreditCard)`, которая показана в листинге 1.4, но в этом случае необходимо определить новый класс `Purchase`. Как вариант, если перед этим вы решили взять `Pair<Donut, Payment>`, вы должны будете использовать `Pair <List<Donut>, Payment>`.

Листинг 1.4. Покупка сразу нескольких пончиков

```
package com.fpinkotlin.introduction.listing05
data class Purchase(val donuts: List<Donut>, val payment: Payment)
fun buyDonuts(quantity: Int = 1, creditCard: CreditCard): Purchase =
    Purchase(List(quantity) {
        Donut()
    }, Payment(creditCard, Donut.price * quantity))
```

Здесь `List(quantity) { Donut() }` создает список с `quantity` элементами и последовательно применяет функцию `{ Donut() }` к значениям от `0` до `quantity - 1`. Функция `{ Donut() }` эквивалентна

```
{ index -> Donut{} }
```

или

```
{ _ -> Donut{} }
```

В случае с единственным параметром часть `parameter ->` можно опустить и ссылаться на параметр как `it`. Поскольку в данном случае он не используется, код сокращается до `{ Donut () }`. Если вам что-то пока не понятно, не беспокойтесь: я расскажу об этом подробнее в следующей главе.

Также обратите внимание, что параметр `quantity` имеет значение по умолчанию `1`. Это позволяет вызвать функцию `buyDonuts`, используя следующий синтаксис без параметра `quantity`:

```
buyDonuts(creditCard = cc)
```

Чтобы реализовать подобную возможность в языке Java, вам пришлось бы написать перегруженную версию метода, например:

```
public static Purchase buyDonuts(CreditCard creditCard) {
    return buyDonuts(1, creditCard);
```

```
}
public static Purchase buyDonuts(int quantity,
                                 CreditCard creditCard) {
    return new Purchase(Collections.nCopies(quantity, new Donut()),
                        new Payment(creditCard, Donut.price * quantity));
}
```

Теперь программу можно протестировать без использования имитаций. Например, вот как мог бы выглядеть тест для метода `buyDonuts`:

```
import org.junit.Assert.assertEquals
import org.junit.Test
class DonutShopKtTest {
    @Test
    fun testBuyDonuts() {
        val creditCard = CreditCard()
        val purchase = buyDonuts(5, creditCard)
        assertEquals(Donut.price * 5, purchase.payment.amount)
        assertEquals(creditCard, purchase.payment.creditCard)
    }
}
```

Еще одно преимущество такой организации кода: программа упрощает комбинирование ее элементов. При совершении сразу нескольких покупок первая версия программы обратится в банк и оплатит каждую покупку отдельно (что повлечет списание банком соответствующей комиссии за каждую покупку). Новая версия может сгруппировать все платежи, сделанные с помощью одной и той же карты, и списать деньги только один раз – сразу за все покупки. Для группировки платежей вам понадобятся дополнительные функции из класса `List` в языке Kotlin:

- `groupBy(f: (A) -> B): Map<B, List<A>>` – принимает функцию, получающую значение типа `A` и возвращающую значение типа `B`, и в свою очередь возвращает ассоциативный массив с ключами типа `B` и значениями типа `List<A>`. Эту функции можно использовать для группировки платежей по кредитной карте.
- `values: List<A>` – функция экземпляра типа `Map`, которая возвращает список всех значений в ассоциативном массиве.
- `map(f: (A) -> B): List<B>` – функция экземпляра типа `List`, которая принимает функцию, получающую значение типа `A` и возвращающую значение типа `B`, применяет ее ко всем элементам в списке `A` и возвращает список элементов типа `B`.
- `reduce(f: (A, A) -> A): A` – функция экземпляра типа `List`, которая использует операцию (представленную функцией `f: (A, A) -> A`) для преобразования (свертки) списка в единственное значение. Операцией может быть, например, сложение. В таком случае можно было бы передать функцию `f(a, b) = a + b`.

Используя эти функции, можно создать новую, которая группирует платежи кредитной картой, как показано в листинге 1.5.

Листинг 1.5. Группировка платежей кредитной картой

```
package com.fpinkotlin.introduction.listing05;
class Payment(val creditCard: CreditCard, val amount: Int) {
    fun combine(payment: Payment): Payment =
        if (creditCard == payment.creditCard)
            Payment(creditCard, amount + payment.amount)
        else
            throw IllegalStateException("Cards don't match.")
    companion object {
        fun groupByCard(payments: List<Payment>): List<Payment> =
            payments.groupBy { it.creditCard } ①
                    .values ②
                    .map { it.reduce(Payment::combine) } ③
    }
}
```

① Преобразует `List<Payment>` в `Map<CreditCard, List<Payment>>`, где каждый список `List<Payment>` содержит все платежи для конкретной кредитной карты
② Преобразует `Map<CreditCard, List<Payment>>` в `List<List<Payment>>`
③ Преобразует список `List<Payment>` в единственный объект `Payment`, содержащий общую сумму всех платежей из `List<Payment>`

Обратите внимание, что в последней строке в функции `groupByCard` используется ссылка на функцию `Payment::combine`. Ссылки на функции подобны ссылкам на методы в Java. Если этот пример не совсем понятен вам, не волнуйтесь – эта книга как раз и предназначена для разъяснения подобных особенностей! Когда дойдете до конца, вы станете экспертом в написании подобного кода.

1.2.3 Максимальное использование абстракций

Как было показано выше, используя *чистые функции*, т. е. функции без побочных эффектов, можно писать более безопасные программы, которые проще тестировать. Объявлять такие функции можно с помощью ключевого слова `fun` или как функции-значения, подобно аргументам методов `groupBy`, `map` или `lower` в листинге 1.5. *Функции-значения* – это такие функции, которые, в отличие от функций `fun`, могут передаваться из одной части программы в другую. Их можно предавать в аргументах другим функциям или возвращать из функций. Как это сделать, вы узнаете в следующих главах.

Но самая важная идея – *абстракция*. Посмотрите на функцию `reduce`. Она принимает аргумент, описывающий операцию, и использует эту операцию, чтобы сократить список до одного значения. Это может быть любая операция, при условии, что она принимает два операнда одного типа.

Рассмотрим список целых чисел. Вы можете написать функцию `sum`, вычисляющую сумму элементов, функцию `product`, вычисляющую произведение элементов, или функцию `min` или `max` для вычисления минимального или максимального элемента списка. Но есть другой путь – использовать для всех этих вычислений функцию `reduce`. Это и есть абстракция. Вы абстрагируете часть, которая является общей для всех операций

в функции `reduce`, и определяете переменную часть (операцию) в виде аргумента.

Можно пойти еще дальше. Функция `reduce` является частным случаем более общей функции, которая может давать результат другого типа, отличного от типа элементов списка. Например, ее можно применить к списку символов и создать строку `String`. Вычисления начинаются с некоторого начального значения (возможно, с пустой строки). В главах 3 и 5 вы узнаете, как использовать эту функцию, называемую `fold`.

Функция `reduce` не будет работать с пустым списком. Представьте список целых чисел: чтобы вычислить сумму, нужно иметь хотя бы один элемент в списке. Но если список пустой, что вернуть в этом случае? Логично предположить, что результатом должен быть 0, но этот подход работает только для вычисления суммы. Он не годится для вычисления произведения.

Вернемся к нашей функции `groupByCard`. Она похожа на бизнес-функцию, которую можно использовать только для группировки платежей по кредитным картам. Но на самом деле это не так! Эту функцию можно использовать для группировки элементов любого списка по любому из их свойств. Эту функцию следовало бы абстрагировать и поместить в класс `List`, чтобы ее можно было использовать повторно. (Она уже определена в Kotlin-классе `List`.)

Максимальное использование абстракций позволяет делать программы еще более безопасными, потому что абстрагированная часть будет написана только один раз. Как следствие, после ее полного тестирования исчезнет риск появления новых ошибок в этой части.

В оставшейся части этой книги вы узнаете множество приемов абстрагирования. Например, как абстрагировать циклы, чтобы избавиться от необходимости писать их снова и снова, как абстрагировать параллельные вычисления, чтобы иметь возможность легко переключаться с последовательной обработки на параллельную, просто выбирая функцию в классе `List`.

Итоги

- Программы можно сделать более безопасными, явно отделяя функции, которые возвращают значения, от эффектов, связанных с взаимодействием с внешним миром.
- Функции легче анализировать и тестировать, потому что их результат предопределен и не зависит от внешнего состояния.
- Максимальное использование абстракций повышает безопасность, удобство сопровождения и простоту тестирования, а также упрощает возможность повторного использования.
- Применение принципов безопасного программирования, таких как неизменяемость и ссылочная прозрачность, защищает программы от случайного использования общего изменяемого состояния, которое часто является источником ошибок в многопоточных окружениях.