

Оглавление

Предисловие.....	19
Благодарности.....	21
Рауль-Габриэль Урма.....	22
Марио Фуско	22
Алан Майкрофт	22
Об этой книге	23
Структура издания	25
О коде.....	27
Форум для обсуждения книги	28
От издательства	28
Об авторах	29
Иллюстрация на обложке	31

Часть I. Основы

Глава 1. Java 8, 9, 10 и 11: что происходит?	35
1.1. Итак, о чем вообще речь?.....	35
1.2. Почему Java все еще меняется.....	38
1.2.1. Место языка Java в экосистеме языков программирования	39
1.2.2. Поточковая обработка.....	41
1.2.3. Передача кода в методы и параметризация поведения.....	42
1.2.4. Параллелизм и разделяемые изменяемые данные	43
1.2.5. Язык Java должен развиваться	44
1.3. Функции в Java	45
1.3.1. Методы и лямбда-выражения как полноправные граждане	46
1.3.2. Передача кода: пример	48
1.3.3. От передачи методов к лямбда-выражениям	50

1.4. Потоки данных	50
1.4.1. Многопоточность — трудная задача	52
1.5. Методы с реализацией по умолчанию и модули Java	55
1.6. Другие удачные идеи, заимствованные из функционального программирования	57
Резюме.....	59
Глава 2. Передача кода и параметризация поведения	60
2.1. Как адаптироваться к меняющимся требованиям.....	61
2.1.1. Первая попытка: фильтрация зеленых яблок.....	61
2.1.2. Вторая попытка: параметризация цвета	62
2.1.3. Третья попытка: фильтрация по всем возможным атрибутам	63
2.2. Параметризация поведения	64
2.2.1. Четвертая попытка: фильтрация по абстрактному критерию	65
2.3. Делаем код лаконичнее.....	69
2.3.1. Анонимные классы.....	70
2.3.2. Пятая попытка: использование анонимного класса.....	70
2.3.3. Шестая попытка: использование лямбда-выражения.....	72
2.3.4. Седьмая попытка: абстрагирование по типу значений списка.....	72
2.4. Примеры из практики	73
2.4.1. Сортировка с помощью интерфейса Comparator	73
2.4.2. Выполнение блока кода с помощью интерфейса Runnable	74
2.4.3. Возвращение результатов выполнения задачи с помощью интерфейса Callable	75
2.4.4. Обработка событий графического интерфейса	75
Резюме.....	76
Глава 3. Лямбда-выражения	77
3.1. Главное о лямбда-выражениях	78
3.2. Где и как использовать лямбда-выражения	81
3.2.1. Функциональный интерфейс.....	81
3.2.2. Функциональные дескрипторы.....	83
3.3. Практическое использование лямбда-выражений: паттерн охватывающего выполнения	85
3.3.1. Шаг 1: вспоминаем параметризацию поведения	86
3.3.2. Шаг 2: используем функциональный интерфейс для передачи поведения	86
3.3.3. Шаг 3: выполняем нужный вариант поведения!.....	87
3.3.4. Шаг 4: передаем лямбда-выражения	87
3.4. Использование функциональных интерфейсов	88
3.4.1. Интерфейс Predicate	89
3.4.2. Интерфейс Consumer	89
3.4.3. Интерфейс Function	90
3.5. Проверка типов, вывод типов и ограничения.....	94
3.5.1. Проверка типов	94
3.5.2. То же лямбда-выражение, другие функциональные интерфейсы.....	96

3.5.3. Вывод типов	98
3.5.4. Использование локальных переменных	98
3.6. Ссылки на методы	100
3.6.1. В общих чертах	100
3.6.2. Ссылки на конструкторы.....	103
3.7. Практическое использование лямбда-выражений и ссылок на методы	106
3.7.1. Шаг 1: передаем код.....	106
3.7.2. Шаг 2: используем анонимный класс	106
3.7.3. Шаг 3: используем лямбда-выражения	106
3.7.4. Шаг 4: используем ссылки на методы	107
3.8. Методы, удобные для композиции лямбда-выражений.....	107
3.8.1. Композиция объектов Comparator	108
3.8.2. Композиция предикатов	109
3.8.3. Композиция функций.....	109
3.9. Схожие математические идеи	111
3.9.1. Интегрирование	111
3.9.2. Переводим на язык лямбда-выражений Java 8.....	112
Резюме.....	114

Часть II. Функциональное программирование с помощью потоков

Глава 4. Знакомство с потоками данных	117
4.1. Что такое потоки данных.....	118
4.2. Знакомимся с потоками данных	122
4.3. Потоки данных и коллекции	125
4.3.1. Строго однократный проход	126
4.3.2. Внутренняя и внешняя итерация	127
4.4. Потоквые операции	130
4.4.1. Промежуточные операции	130
4.4.2. Завершающие операции	131
4.4.3. Работа с потоками данных.....	132
Резюме.....	133
Глава 5. Работа с потоками данных	134
5.1. Фильтрация	135
5.1.1. Фильтрация с помощью предиката	135
5.1.2. Фильтрация уникальных элементов	136
5.2. Срез потока данных.....	136
5.2.1. Срез с помощью предиката.....	137
5.2.2. Усечение потока данных.....	138
5.2.3. Пропуск элементов	139
5.3. Отображение.....	140
5.3.1. Применение функции к каждому из элементов потока данных	140
5.3.2. Схлопывание потоков данных.....	141

5.4. Поиск и сопоставление с шаблоном	144
5.4.1. Проверяем, удовлетворяет ли предикату хоть один элемент	144
5.4.2. Проверяем, удовлетворяют ли предикату все элементы	145
5.4.3. Поиск элемента в потоке	145
5.4.4. Поиск первого элемента	146
5.5. Свертка	147
5.5.1. Суммирование элементов	147
5.5.2. Максимум и минимум	149
5.6. Переходим к практике	153
5.6.1. Предметная область: трейдеры и транзакции	154
5.6.2. Решения	155
5.7. Числовые потоки данных	157
5.7.1. Версии потоков для простых типов данных	157
5.7.2. Числовые диапазоны	159
5.7.3. Применяем числовые потоки данных на практике: пифагоровы тройки	159
5.8. Создание потоков данных	162
5.8.1. Создание потока данных из перечня значений	162
5.8.2. Создание потока данных из объекта, допускающего неопределенное значение	163
5.8.3. Создание потоков данных из массивов	163
5.8.4. Создание потоков данных из файлов	163
5.8.5. Потоки на основе функций: создание бесконечных потоков	164
Резюме	169
Глава 6. Сбор данных с помощью потоков	170
6.1. Главное о коллекторах	172
6.1.1. Коллекторы как продвинутое средства свертки	172
6.1.2. Встроенные коллекторы	173
6.2. Свертка и вычисление сводных показателей	174
6.2.1. Поиск максимума и минимума в потоке данных	174
6.2.2. Вычисление сводных показателей	175
6.2.3. Объединение строк	176
6.2.4. Обобщение вычисления сводных показателей с помощью свертки	177
6.3. Группировка	181
6.3.1. Дальнейшие операции со сгруппированными элементами	183
6.3.2. Многоуровневая группировка	184
6.3.3. Сбор данных в подгруппы	186
6.4. Секционирование	190
6.4.1. Преимущества секционирования	190
6.4.2. Секционирование чисел на простые и составные	192
6.5. Интерфейс Collector	194
6.5.1. Разбираемся с объявленными в интерфейсе Collector методами	195
6.5.2. Собираем все вместе	199

6.6. Разрабатываем собственный, более производительный коллектор	201
6.6.1. Деление только на простые числа	201
6.6.2. Сравнение производительности коллекторов	205
Резюме.....	207
Глава 7. Параллельная обработка данных и производительность	208
7.1. Параллельные потоки данных.....	209
7.1.1. Превращаем последовательный поток данных в параллельный	210
7.1.2. Измерение быстродействия потока данных.....	212
7.1.3. Правильное применение параллельных потоков данных.....	217
7.1.4. Эффективное использование параллельных потоков данных.....	218
7.2. Фреймворк ветвления-объединения.....	220
7.2.1. Работа с классом RecursiveTask	220
7.2.2. Рекомендуемые практики применения фреймворка ветвления- объединения.....	224
7.2.3. Перехват работы	225
7.3. Интерфейс Spliterator	226
7.3.1. Процесс разбиения	227
7.3.2. Реализация собственного сплитератора.....	229
Резюме.....	234

Часть III. Эффективное программирование с помощью потоков и лямбда-выражений

Глава 8. Расширения Collection API	237
8.1. Фабрики коллекций	238
8.1.1. Фабрика списков	239
8.1.2. Фабрика множеств.....	240
8.1.3. Фабрики ассоциативных массивов	240
8.2. Работа со списками и множествами	241
8.2.1. Метод removeIf.....	242
8.2.2. Метод replaceAll	243
8.3. Работа с ассоциативными массивами	244
8.3.1. Метод forEach	244
8.3.2. Сортировка.....	244
8.3.3. Метод getOrDefault	245
8.3.4. Паттерны вычисления	246
8.3.5. Паттерны удаления	247
8.3.6. Способы замены элементов	248
8.3.7. Метод merge	248
8.4. Усовершенствованный класс ConcurrentHashMap	250
8.4.1. Свертка и поиск.....	250
8.4.2. Счетчики	251
8.4.3. Представление в виде множества.....	252
Резюме.....	252

Глава 9. Рефакторинг, тестирование и отладка	253
9.1. Рефакторинг с целью повышения удобочитаемости и гибкости кода.....	254
9.1.1. Повышаем удобочитаемость кода.....	254
9.1.2. Из анонимных классов — в лямбда-выражения.....	254
9.1.3. Из лямбда-выражений — в ссылки на методы.....	256
9.1.4. От императивной обработки данных до потоков.....	257
9.1.5. Повышаем гибкость кода.....	258
9.2. Рефакторинг объектно-ориентированных паттернов проектирования с помощью лямбда-выражений	260
9.2.1. Стратегия	261
9.2.2. Шаблонный метод	263
9.2.3. Наблюдатель	264
9.2.4. Цепочка обязанностей.....	266
9.2.5. Фабрика	268
9.3. Тестирование лямбда-выражений.....	270
9.3.1. Тестирование поведения видимого лямбда-выражения	270
9.3.2. Делаем упор на поведение метода, использующего лямбда-выражение	271
9.3.3. Разносим сложные лямбда-выражения по отдельным методам	272
9.3.4. Тестирование функций высшего порядка.....	272
9.4. Отладка.....	272
9.4.1. Изучаем трассу вызовов в стеке	273
9.4.2. Журналирование информации.....	274
Резюме.....	276
Глава 10. Предметно-ориентированные языки и лямбда-выражения	277
10.1. Специальный язык для конкретной предметной области.....	279
10.1.1. За и против предметно-ориентированных языков	280
10.1.2. Доступные на JVM решения, подходящие для создания DSL	282
10.2. Малые DSL в современных API Java	286
10.2.1. Stream API как DSL для работы с коллекциями	287
10.2.2. Коллекторы как предметно-ориентированный язык агрегирования данных	289
10.3. Паттерны и методики создания DSL на языке Java.....	290
10.3.1. Связывание методов цепочкой	293
10.3.2. Вложенные функции.....	295
10.3.3. Задание последовательности функций с помощью лямбда-выражений	297
10.3.4. Собираем все воедино	300
10.3.5. Использование ссылок на методы в DSL	302
10.4. Реальные примеры DSL Java 8	304
10.4.1. jOOQ.....	305
10.4.2. Cucumber	306
10.4.3. Фреймворк Spring Integration	308
Резюме.....	310

Часть IV. Java на каждый день

Глава 11. Класс Optional как лучшая альтернатива null	313
11.1. Как смоделировать отсутствие значения.....	314
11.1.1. Снижение количества исключений NullPointerException с помощью проверки на безопасность	315
11.1.2. Проблемы, возникающие с null	316
11.1.3. Альтернативы null в других языках программирования	316
11.2. Знакомство с классом Optional	318
11.3. Паттерны для внедрения в базу кода опционалов.....	319
11.3.1. Создание объектов Optional	319
11.3.2. Извлечение и преобразование значений опционалов с помощью метода map	320
11.3.3. Связывание цепочкой объектов Optional с помощью метода flatMap	321
11.3.4. Операции над потоком опционалов	325
11.3.5. Действия по умолчанию и распаковка опционалов	326
11.3.6. Сочетание двух опционалов.....	327
11.3.7. Отбрасывание определенных значений с помощью метода filter	328
11.4. Примеры использования опционалов на практике	330
11.4.1. Обертывание потенциально пустого значения в опционал.....	331
11.4.2. Исключения или опционалы?	331
11.4.3. Версии опционалов для простых типов данных и почему их лучше не использовать	332
11.4.4. Собираем все воедино	332
Резюме.....	334
Глава 12. Новый API для работы с датой и временем	335
12.1. Классы LocalDate, LocalTime, LocalDateTime, Instant, Duration и Period.....	336
12.1.1. Работа с классами LocalDate и LocalTime.....	336
12.1.2. Сочетание даты и времени	338
12.1.3. Класс Instant: дата и время для компьютеров	338
12.1.4. Описание продолжительности и промежутков времени	339
12.2. Операции над датами, синтаксический разбор и форматирование дат.....	341
12.2.1. Работа с классом TemporalAdjusters.....	343
12.2.2. Вывод в консоль и синтаксический разбор объектов даты/времени	345
12.3. Различные часовые пояса и системы летоисчисления	347
12.3.1. Использование часовых поясов.....	347
12.3.2. Фиксированное смещение от UTC/времени по Гринвичу	348
12.3.3. Использование альтернативных систем летоисчисления.....	349
Резюме.....	350

Глава 13. Методы с реализацией по умолчанию	351
13.1. Эволюция API	354
13.1.1. Версия 1 API	355
13.1.2. Версия 2 API	355
13.2. Коротко о методах с реализацией по умолчанию	357
13.3. Примеры использования методов с реализацией по умолчанию	360
13.3.1. Необязательные методы	360
13.3.2. Множественное наследование поведения	360
13.4. Правила разрешения конфликтов	364
13.4.1. Три важных правила разрешения неоднозначностей	365
13.4.2. Преимущество — у самого конкретного интерфейса из числа содержащих реализацию по умолчанию	365
13.4.3. Конфликты и разрешение неоднозначностей явным образом.....	367
13.4.4. Проблема ромбовидного наследования.....	368
Резюме.....	370
Глава 14. Система модулей Java	371
14.1. Движущая сила появления системы модулей Java: размышления о ПО	372
14.1.1. Разделение ответственности.....	372
14.1.2. Скрытие информации	372
14.1.3. Программное обеспечение на языке Java.....	373
14.2. Причины создания системы модулей Java	374
14.2.1. Ограничения модульной организации	374
14.2.2. Монолитный JDK.....	376
14.2.3. Сравнение с OSGi.....	376
14.3. Модули Java: общая картина.....	377
14.4. Разработка приложения с помощью системы модулей Java.....	378
14.4.1. Подготовка приложения	378
14.4.2. Мелкоблочная и крупноблочная модуляризация	381
14.4.3. Основы системы модулей Java	381
14.5. Работа с несколькими модулями	382
14.5.1. Выражение exports	383
14.5.1. Выражение requires.....	383
14.5.3. Именованное	384
14.6. Компиляция и компоновка	385
14.7. Автоматические модули	388
14.8. Объявление модуля и выражения	389
14.8.1. Выражение requires.....	389
14.8.2. Выражение exports	389
14.8.3. Выражение requires transitive	390
14.8.4. Выражение exports ... to.....	390
14.8.5. Выражения open и opens	390
14.8.6. uses и provides	391
14.9. Пример побольше и дополнительные источники информации	391
Резюме.....	392

Часть V. Расширенная конкурентность в языке Java

Глава 15. Основные концепции класса <code>CompletableFuture</code> и реактивное программирование	395
15.1. Эволюция поддержки конкурентности в Java	398
15.1.1. Потоки выполнения и высокоуровневые абстракции.....	399
15.1.2. Исполнители и пулы потоков выполнения	401
15.1.3. Другие абстракции потоков выполнения: (не) вложенность в вызовы методов.....	403
15.1.4. Что нам нужно от потоков выполнения.....	405
15.2. Синхронные и асинхронные API	406
15.2.1. Фьючерсный API	408
15.2.2. Реактивный API	408
15.2.3. Приостановка и другие блокирующие операции вредны	410
15.2.4. Смотрим правде в глаза.....	412
15.2.5. Исключения и асинхронные API	412
15.3. Модель блоков и каналов	413
15.4. Реализация конкурентности с помощью класса <code>CompletableFuture</code> и комбинаторов	415
15.5. Публикация/подписка и реактивное программирование.....	419
15.5.1. Пример использования для суммирования двух потоков сообщений.....	420
15.5.2. Противодействие.....	425
15.5.3. Простой вариант реализации противодействия на практике.....	425
15.6. Реактивные системы и реактивное программирование.....	426
Резюме.....	427
Глава 16. Класс <code>CompletableFuture</code> : композиция асинхронных компонентов	428
16.1. Простые применения фьючерсов	429
16.1.1. Разбираемся в работе фьючерсов и их ограничениях	430
16.1.2. Построение асинхронного приложения с помощью завершаемых фьючерсов	431
16.2. Реализация асинхронного API	432
16.2.1. Преобразование синхронного метода в асинхронный.....	433
16.2.2. Обработка ошибок.....	435
16.3. Делаем код неблокирующим.....	437
16.3.1. Распараллеливание запросов с помощью параллельного потока данных.....	438
16.3.2. Выполнение асинхронных запросов с помощью завершаемых фьючерсов	438
16.3.3. Поиск лучше масштабируемого решения	441
16.3.4. Пользовательский исполнитель	442
16.4. Конвейерная организация асинхронных задач	444
16.4.1. Реализация сервиса скидок.....	444
16.4.2. Использование скидочного сервиса	446
16.4.3. Композиция синхронных и асинхронных операций.....	446

16.4.4. Сочетание двух завершаемых фьючерсов: зависимого и независимого	449
16.4.5. Сравниваем фьючерсы с завершаемыми фьючерсами.....	450
16.4.6. Эффективное использование тайм-аутов	452
16.5. Реагирование на завершение CompletableFuture	453
16.5.1. Рефакторинг приложения для поиска наилучшей цены	454
16.5.2. Собираем все вместе	455
Резюме.....	456
Глава 17. Реактивное программирование	457
17.1. Манифест реактивности.....	458
17.1.1. Реактивность на прикладном уровне.....	459
17.1.2. Реактивность на системном уровне	461
17.2. Реактивные потоки данных и Flow API	462
17.2.1. Знакомство с классом Flow.....	463
17.2.2. Создаем ваше первое реактивное приложение	466
17.2.3. Преобразование данных с помощью интерфейса Processor	471
17.2.4. Почему Java не предоставляет реализации Flow API	473
17.3. Реактивная библиотека RxJava.....	474
17.3.1. Создание и использование наблюдаемых объектов.....	475
17.3.2. Преобразование и группировка наблюдаемых объектов	480
Резюме.....	484
Часть VI. Функциональное программирование и эволюция языка Java	
Глава 18. Мыслим функционально	487
18.1. Создание и сопровождение систем	488
18.1.1. Разделяемые изменяемые данные	488
18.1.2. Декларативное программирование	490
18.1.3. Почему функциональное программирование?	491
18.2. Что такое функциональное программирование	491
18.2.1. Функциональное программирование на языке Java.....	492
18.2.2. Функциональная прозрачность	494
18.2.3. Объектно-ориентированное и функциональное программирование	495
18.2.4. Функциональное программирование в действии	496
18.3. Рекурсия и итерация.....	498
Резюме.....	502
Глава 19. Методики функционального программирования	503
19.1. Повсюду функции	503
19.1.1. Функции высшего порядка.....	504
19.1.2. Каррирование.....	506
19.2. Персистентные структуры данных.....	507
19.2.1. Деструктивные и функциональные обновления	508

19.2.2. Другой пример, с деревьями.....	510
19.2.3. Функциональный подход.....	511
19.3. Отложенное вычисление с помощью потоков данных	513
19.3.1. Самоопределяющиеся потоки данных.....	513
19.3.2. Наш собственный отложенный список	516
19.4. Сопоставление с шаблоном.....	521
19.4.1. Паттерн проектирования «Посетитель»	521
19.4.2. На помощь приходит сопоставление с шаблоном	522
19.5. Разное	525
19.5.1. Кэширование или мемоизация	525
19.5.2. Что значит «вернуть тот же самый объект»?	527
19.5.3. Комбинаторы	527
Резюме.....	529
Глава 20. Смесь ООП и ФП: сравнение Java и Scala	530
20.1. Введение в Scala.....	531
20.1.1. Приложение Hello beer.....	531
20.1.2. Основные структуры данных Scala: List, Set, Map, Stream, Tuple и Option	533
20.2. Функции	539
20.2.1. Полноправные функции.....	539
20.2.2. Анонимные функции и замыкания.....	540
20.2.3. Каррирование.....	542
20.3. Классы и типы.....	543
20.3.1. Код с классами Scala становится лаконичнее	543
20.3.2. Типы Scala и интерфейсы Java	545
Резюме.....	546
Глава 21. Заключение и дальнейшие перспективы Java	547
21.1. Обзор возможностей Java 8	547
21.1.1. Параметризация поведения (лямбда-выражения и ссылки на методы).....	548
21.1.2. Потоки данных.....	549
21.1.3. Класс CompletableFuture.....	549
21.1.4. Класс Optional	550
21.1.5. Flow API.....	551
21.1.6. Методы с реализацией по умолчанию.....	551
21.2. Система модулей Java 9	551
21.3. Вывод типов локальных переменных в Java 10	553
21.4. Что ждет Java в будущем?.....	554
21.4.1. Вариантность по месту объявления	554
21.4.2. Сопоставление с шаблоном.....	555
21.4.3. Более полнофункциональные формы обобщенных типов.....	556

21.4.4. Расширение поддержки неизменяемости	558
21.4.5. Типы-значения	559
21.5. Ускорение развития Java.....	562
21.6. Заключительное слово.....	564

Приложения

Приложение А. Прочие изменения языка	566
А.1. Аннотации	566
А.1.1. Повторяющиеся аннотации	567
А.1.2. Аннотации типов	568
А.2. Обобщенный вывод целевых типов	569
Приложение Б. Прочие изменения в библиотеках	570
Б.1. Коллекции	570
Б.1.1. Добавленные методы	570
Б.1.2. Класс Collections	572
Б.1.3. Интерфейс Comparator	572
Б.2. Конкурентность	572
Б.2.1. Пакет Atomic	573
Б.2.2. ConcurrentHashMap.....	574
Б.3. Массивы	575
Б.3.1. Использование метода parallelSort	575
Б.3.2. Использование методов setAll и parallelSetAll	576
Б.3.3. Использование метода parallelPrefix.....	576
Б.4. Классы Number и Math.....	576
Б.4.1. Класс Number	576
Б.4.2. Класс Math	577
Б.5. Класс Files	577
Б.6. Рефлексия	577
Б.7. Класс String	578
Приложение В. Параллельное выполнение нескольких операций над потоком данных	579
В.1. Ветвление потока данных.....	580
В.1.1. Реализация интерфейса Results на основе класса ForkingStreamConsumer	581
В.1.2. Разработка классов ForkingStreamConsumer и BlockingQueueSpliterator	583
В.1.3. Применяем StreamForker на практике	585
В.2. Вопросы производительности	587
Приложение Г. Лямбда-выражения и байт-код JVM	588
Г.1. Анонимные классы.....	588
Г.2. Генерация байт-кода	589
Г.3. Invokedynamic спешит на помощь	590
Г.4. Стратегии генерации кода	591

1.3.3. От передачи методов к лямбда-выражениям

Передавать методы как значения, безусловно, удобно, хотя весьма досаждают необходимость описывать даже короткие методы, такие как `isHeavyApple` и `isGreenApple`, которые используются только один-два раза. Но Java 8 решает и эту проблему. В нем появилась новая нотация (анонимные функции, называемые также лямбда-выражениями), благодаря которой можно написать просто:

```
filterApples(inventory, (Apple a) -> GREEN.equals(a.getColor()) );
```

или:

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

или даже:

```
filterApples(inventory, (Apple a) -> a.getWeight() < 80 ||
              RED.equals(a.getColor()) );
```

Используемый однократно метод не требуется описывать; код становится четче и понятнее, поскольку не нужно тратить время на поиски по исходному коду, чтобы понять, какой именно код передается.

Но если размер подобного лямбда-выражения превышает несколько строк (и его поведение сразу не понятно), лучше воспользоваться вместо анонимной лямбды ссылкой на метод с наглядным именем. Понятность кода — превыше всего.

Создатели Java 8 могли на этом практически завершить свою работу и, наверное, так бы и поступили, если бы не многоядерные процессоры. Как вы увидите, функциональное программирование в представленном нами объеме обладает весьма широкими возможностями. Язык Java мог бы в качестве вишенки на торте добавить обобщенный библиотечный метод `filter` и несколько родственных ему, например:

```
static <T> Collection<T> filter(Collection<T> c, Predicate<T> p);
```

Вам не пришлось бы даже писать самим такие методы, как `filterApples`, поскольку даже предыдущий вызов:

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

можно было бы переписать в виде обращения к библиотечному методу `filter`:

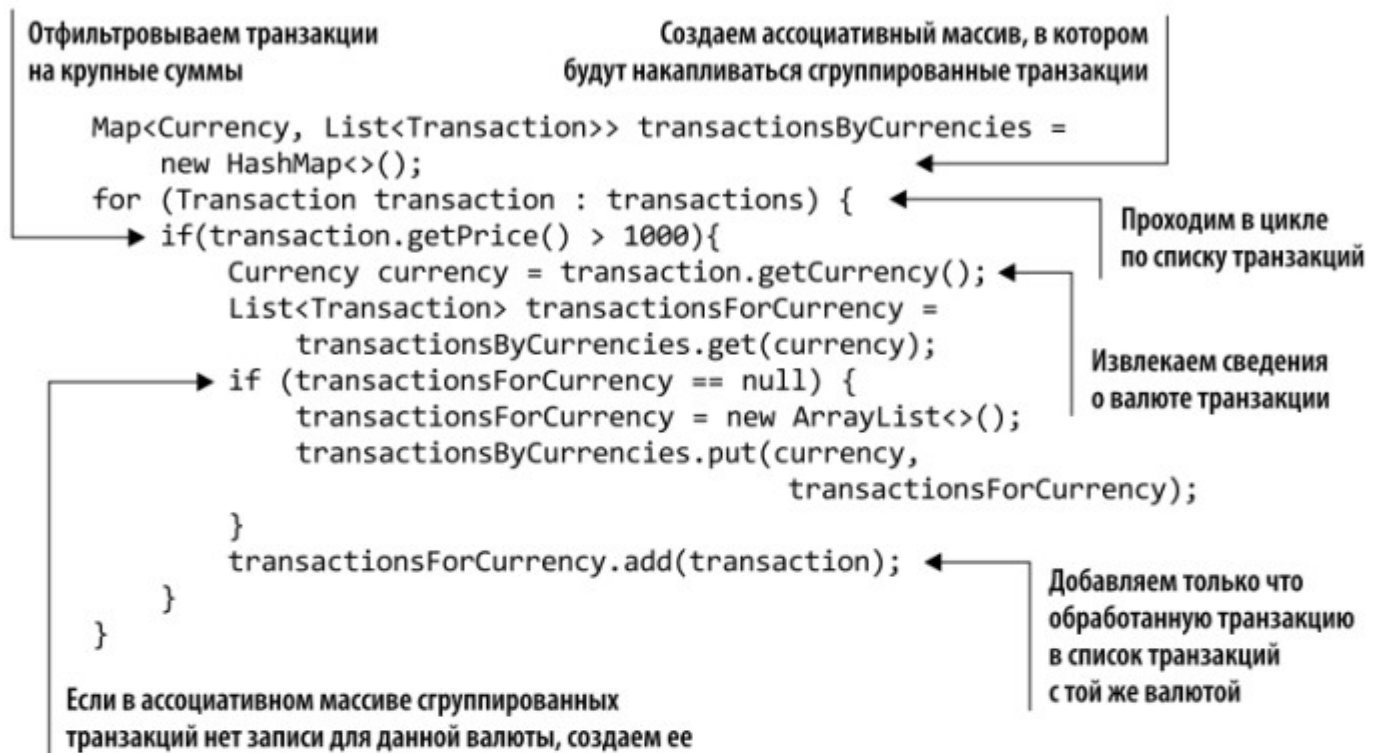
```
filter(inventory, (Apple a) -> a.getWeight() > 150 );
```

Но из соображений оптимального использования параллелизма создатели Java не пошли по этому пути. Вместо этого Java 8 включает новый Stream API (поток данных, сходных с коллекциями), содержащий обширный набор подобных `filter`-операций, привычных функциональным программистам (например, `map` и `reduce`), а также методы для преобразования коллекций в потоки данных и наоборот. Этот API мы сейчас и обсудим.

1.4. Потоки данных

Практически любое приложение Java *создает* и *обрабатывает* коллекции. Однако работать с коллекциями не всегда удобно. Например, представьте себе,

что вам нужно отфильтровать из списка транзакции на крупные суммы, а затем сгруппировать их по валюте. Для реализации подобного запроса по обработке данных вам пришлось бы написать огромное количество стереотипного кода, как показано ниже:



Помимо прочего, весьма непросто понять с первого взгляда, что этот код делает, из-за многочисленных вложенных операторов управления потоком выполнения.

С помощью Stream API можно решить ту же задачу следующим образом:

```

List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
    public boolean test(Apple a){
        return RED.equals(a.getColor());
    }
});
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Whoooo a click!!");
    }
}

```

Большое количество стереотипного кода

Не стоит переживать, если этот код пока кажется вам своего рода магией. Главы 4–7 помогут вам разобраться со Stream API. Пока стоит отметить лишь, что этот API позволяет обрабатывать данные несколько иначе, чем Collection API. При использовании коллекции вам приходится организовывать цикл самостоятельно: пройтись по всем элементам, обрабатывая их по очереди в цикле `for-each`. Подобная итерация по данным называется *внешней итерацией* (external iteration). При использовании же Stream API думать о циклах не нужно. Вся обработка данных происходит внутри библиотеки. Мы будем называть это *внутренней итерацией* (internal iteration).

Вторая основная проблема при работе с коллекциями: как обработать список транзакций, если их очень много? Одноядерный процессор не сможет обработать такой большой объем данных, но, вероятно, процессор вашего компьютера — многоядерный. Оптимально было бы разделить объем работ между ядрами процессора, чтобы уменьшить время обработки. Теоретически при наличии восьми ядер обработка данных должна занять в восемь раз меньше времени, поскольку они работают параллельно.

Многоядерные компьютеры

Все новые настольные компьютеры и ноутбуки — многоядерные. Они содержат не один, а несколько процессоров (обычно называемых ядрами). Проблема в том, что обычная программа на языке Java использует лишь одно из этих ядер, а остальные простаивают. Аналогично во многих компаниях для эффективной обработки больших объемов данных применяются *вычислительные кластеры* (computing clusters) — компьютеры, соединенные быстрыми сетями. Java 8 продвигает новые стили программирования, чтобы лучше использовать подобные компьютеры.

Поисковый движок Google — пример кода, который слишком велик для работы на отдельной машине. Он читает все страницы в Интернете и создает индекс соответствия всех встречающихся на каждой странице слов адресу (URL) этой страницы. Далее при поиске в Google по нескольким словам ПО может воспользоваться этим индексом для быстрой выдачи набора страниц, содержащих эти слова. Попробуйте представить себе реализацию этого алгоритма на языке Java (даже в случае гораздо меньшего индекса, чем у Google, вам придется использовать все процессорные ядра вашего компьютера).

1.4.1. Многопоточность — трудная задача

Проблема в том, что реализовать параллелизм, прибегнув к *многопоточному* (multithreaded) коду (с помощью Thread API из предыдущих версий Java), — довольно непростая задача. Лучше воспользоваться другим способом, ведь потоки выполнения могут одновременно обращаться к разделяемым переменным и изменять их. В результате данные могут меняться самым неожиданным образом, если их использование не согласовано, как полагается¹. Такая модель сложнее для понимания², чем последовательная, пошаговая модель. Например, одна из возможных проблем с двумя (не синхронизированными должным образом) потоками выполнения, которые пытаются прибавить число к разделяемой переменной `sum`, приведена на рис. 1.5.

¹ Обычно это делают с помощью ключевого слова `synchronized`, но если указать его не там, где нужно, то может возникнуть множество ошибок, которые трудно обнаружить. Параллелизм на основе потоков данных Java 8 поощряет программирование в функциональном стиле, при котором `synchronized` используется редко; упор делается на секционировании данных, а не на координации доступа к ним.

² А вот и одна из причин, побуждающих язык эволюционировать!

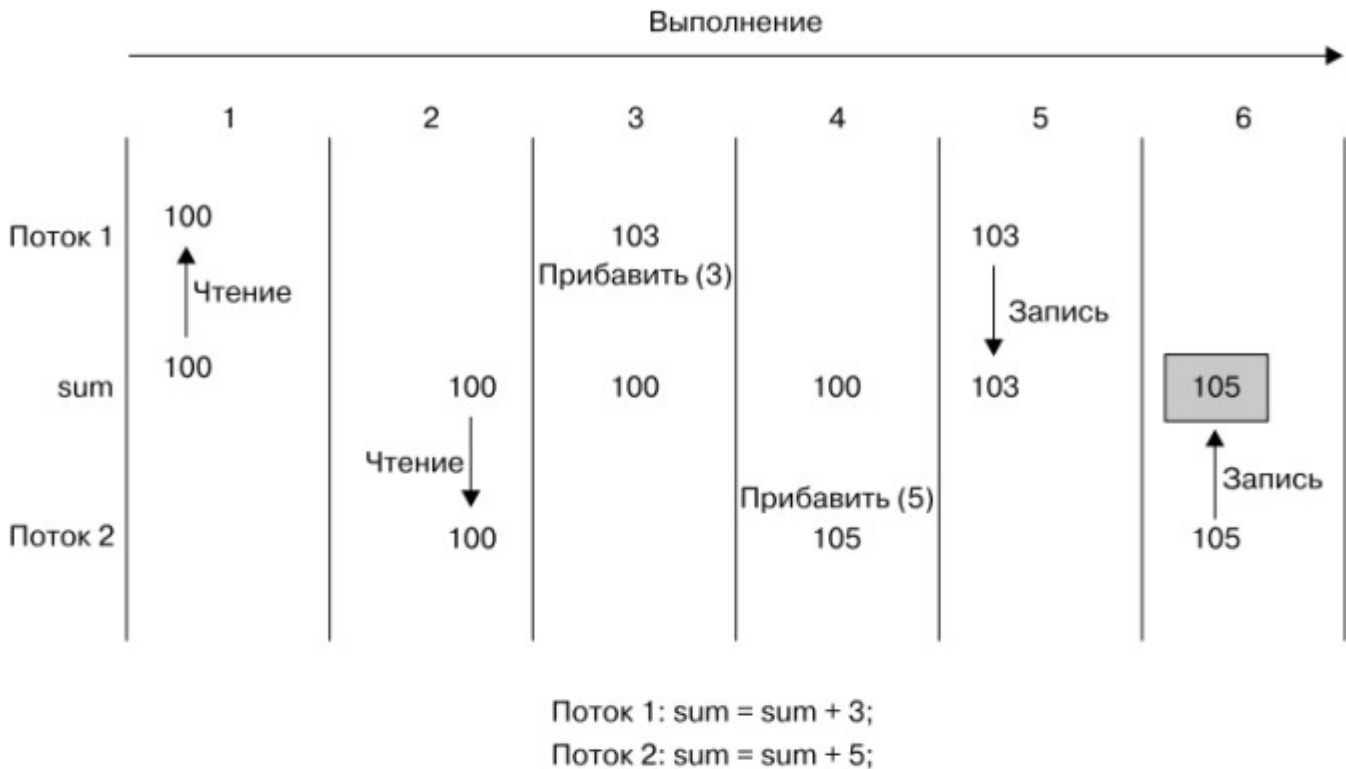


Рис. 1.5. Возможная проблема с двумя потоками выполнения, пытающимися прибавить число к разделяемой переменной `sum`. Результат равен 105 вместо ожидаемых 108

Java 8 решает обе проблемы (стереотипность и малопонятность кода обработки коллекций, а также трудности использования многоядерности) с помощью Stream API (`java.util.stream`). Первый фактор, определивший архитектуру этого API, — существование множества часто встречающихся паттернов обработки данных (таких как `filterApples` из предыдущего раздела или операции, знакомые вам по языкам запросов вроде SQL), которые разумно было бы поместить в библиотеку: *фильтрация* данных по какому-либо критерию (например, выбор тяжелых яблок), *извлечение* (например, извлечение значения поля «вес» из всех яблок в списке) или *группировка* данных (например, группировка списка чисел по отдельным спискам четных и нечетных чисел) и т. д. Второй фактор — часто встречающаяся возможность распараллеливания подобных операций. Например, как показано на рис. 1.6, фильтрацию списка на двух процессорах можно осуществить путем его разбиения на две половины, одну из которых будет обрабатывать один процессор, а вторую — другой. Это называется *шагом ветвления* (forking step) ❶. Далее процессоры фильтруют свои половины списка ❷. И наконец ❸, один из процессоров объединяет результаты (это очень похоже на алгоритм, благодаря которому поиск в Google так быстро работает, хотя там используется гораздо больше двух процессоров).

Пока мы скажем только, что новый Stream API ведет себя аналогично Collection API: оба обеспечивают доступ к последовательностям элементов данных. Но запомните: Collection API в основном связан с хранением и доступом к данным, а Stream API — с описанием производимых над данными вычислений. Важнее всего то, что Stream API позволяет и поощряет параллельную обработку элементов потока. Хотя

на первый взгляд это может показаться странным, но часто самый быстрый способ фильтрации коллекции (например, использовать `filterApples` из предыдущего раздела для списка) — преобразовать ее в поток данных, обработать параллельно и затем преобразовать обратно в список. Мы снова скажем «параллелизм практически даром» и покажем вам, как отфильтровать тяжелые яблоки из списка последовательно и как — параллельно, с помощью потоков данных и лямбда-выражения.

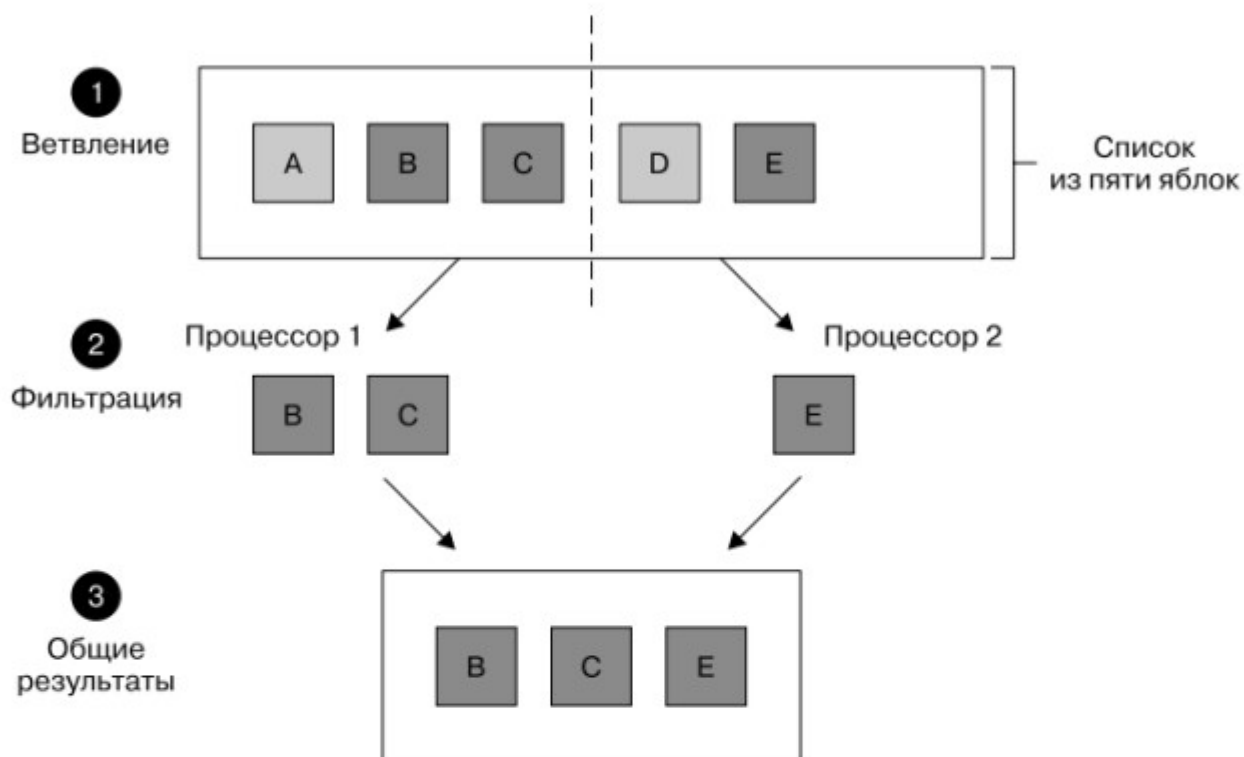


Рис. 1.6. Ветвление `filter` на два процессора и объединение результатов

Вот пример последовательной обработки:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
               .collect(toList());
```

А вот пример параллельной обработки:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
               .collect(toList());
```

Параллелизм в языке Java и отсутствие разделяемого изменяемого состояния

Параллелизм в Java всегда считался непростой задачей, а ключевое слово `synchronized` — источником потенциальных ошибок. Так что же за чудодейственное средство появилось в Java 8?

На самом деле чудодейственных средств два. Во-первых, библиотека, обеспечивающая секционирование — разбиение больших потоков данных на несколько маленьких, для параллельной обработки. Во-вторых, параллелизм «практически даром» с помощью потоков возможен только в том случае, если методы, передаваемые библиотечным методам, таким как `filter`, не взаимодействуют между собой (например, через изменяемые разделяемые объекты). Но оказывается, что это ограничение представляется программистам вполне естественным (см. в качестве примера наш `Apple::isGreenApple`). Хотя основной смысл слова «функциональный» во фразе «функциональное программирование» — «использующий функции в качестве полноценных значений», у него часто есть оттенок «без взаимодействия между компонентами во время выполнения».

В главе 7 вы найдете более подробное обсуждение параллельной обработки данных в Java 8 и вопросов ее производительности. Одна из проблем, с которыми столкнулись на практике разработчики Java, несмотря на все ее замечательные новинки, касалась эволюции существующих интерфейсов. Например, метод `Collections.sort` относится к интерфейсу `List`, но так и не был в него включен. В идеале хотелось бы иметь возможность написать `list.sort(comparator)` вместо `Collections.sort(list, comparator)`. Это может показаться тривиальным, но до Java 8 для обновления интерфейса необходимо было обновить все реализующие его классы — просто логистический кошмар! Данная проблема решена в Java 8 с помощью *методов с реализацией по умолчанию*.

1.5. Методы с реализацией по умолчанию и модули Java

Как мы уже упоминали ранее, современные системы обычно строятся из компонентов — возможно, приобретенных на стороне. Исторически в языке Java не было поддержки этой возможности, за исключением JAR-файлов, хранящих набор Java-пакетов без четкой структуры. Более того, развивать интерфейсы, содержащиеся в таких пакетах, было непросто — изменение Java-интерфейса означало необходимость изменения каждого реализующего его класса. Java 8 и 9 вступили на путь решения этой проблемы.

Во-первых, в Java 9 есть система модулей и синтаксис для описания *модулей*, содержащих наборы пакетов, а контроль видимости и пространств имен значительно усовершенствован. Благодаря модулям у простого компонента типа JAR появляется структура, как для пользовательской документации, так и для автоматической проверки (мы расскажем об этом подробнее в главе 14). Во-вторых, в Java 8 появились методы с реализацией по умолчанию для поддержки *изменяемых интерфейсов*. Мы рассмотрим их подробнее в главе 13. Знать про них важно, ведь они часто будут встречаться вам в интерфейсах, но, поскольку относительно немногим программистам приходится самим писать методы с реализацией по умолчанию, а также поскольку они скорее способствуют эволюции программ, а не

помогают написанию какой-либо конкретной программы, мы расскажем здесь о них кратко, на примере.

В разделе 1.4 приводился следующий пример кода Java 8:

```
List<Apple> heavyApples1 =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                .collect(toList());
List<Apple> heavyApples2 =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                .collect(toList());
```

С этим кодом есть проблема: до Java 8 в интерфейсе `List<T>` не было методов `stream` или `parallelStream` — как и в интерфейсе `Collection<T>`, который он реализует, — поскольку эти методы еще не придумали. А без них такой код не скомпилируется. Простейшее решение для создателей Java 8, какое вы могли бы использовать для своих интерфейсов, — добавить метод `stream` в интерфейс `Collection` и его реализацию в класс `ArrayList`.

Но это стало бы настоящим кошмаром для пользователей, ведь множество разных фреймворков коллекций реализуют интерфейсы из `Collection API`. Добавление нового метода в интерфейс означает необходимость реализации его во всех конкретных классах. У создателей языка нет контроля над существующими реализациями интерфейса `Collection`, так что возникает дилемма: как развивать опубликованные интерфейсы, не нарушая при этом существующие реализации?

Решение, предлагаемое Java 8, заключается в разрыве последнего звена этой цепочки: отныне интерфейсы могут содержать сигнатуры методов, которые не реализуются в классе-реализации. Но кто же тогда их реализует? Отсутствующие тела методов описываются в интерфейсе (поэтому и называются реализациями по умолчанию), а не в реализующем классе.

Благодаря этому у разработчика интерфейса появляется способ увеличить интерфейс, выйдя за пределы изначально планировавшихся методов и не нарушая работу существующего кода. Для этого в Java 8 в спецификациях интерфейсов можно использовать уже существующее ключевое слово `default`.

Например, в Java 8 можно вызвать метод `sort` непосредственно для списка. Это возможно благодаря следующему методу с реализацией по умолчанию из интерфейса `List`, вызывающему статический метод `Collections.sort`:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

Это значит, что конкретные классы интерфейса `List` не обязаны явным образом реализовывать метод `sort`, в то время как в предыдущих версиях Java без такой реализации они бы не скомпилировались.

Но подождите секундочку. Один класс может реализовывать несколько интерфейсов, так? Не будут ли несколько реализаций по умолчанию в нескольких интерфейсах означать возможность своего рода множественного наследования в Java? Да, до некоторой степени. В главе 13 мы обсудим правила, предотвращающие такие проблемы, как печально известная *проблема ромбовидного наследования* (*diamond inheritance problem*) в языке C++.

1.6. Другие удачные идеи, заимствованные из функционального программирования

В предыдущих разделах мы познакомили вас с двумя основными идеями функционального программирования, которые ныне стали частью Java: с использованием методов и лямбда-выражений как полноправных значений, а также с идеей эффективного и безопасного параллельного выполнения функций и методов при условии отсутствия изменяемого разделяемого состояния. Обе эти идеи были воплощены в описанном выше новом Stream API.

В распространенных функциональных языках (SML, OCaml, Haskell) есть и другие конструкции, помогающие программистам в их работе. В их числе возможность обойтись без `null` за счет более явных типов данных. Тони Хоар (Tony Hoare), один из корифеев теории вычислительной техники, сказал на презентации во время конференции QCon в Лондоне в 2009 году:

«Я считаю это своей ошибкой на миллиард долларов: изобретение ссылки на `null` в 1965-м... Я поддался искушению включить в язык ссылку на `null` просто потому, что ее реализация была так проста».

В Java 8 появился класс `Optional<T>`, который при систематическом использовании помогает избегать исключений, связанных с указателем на `null`. Это объект-контейнер, который может содержать или не содержать значение. В `Optional<T>` есть методы для обработки явным образом варианта отсутствия значения, в результате чего можно избежать исключений, связанных с указателем на `null`. С помощью системы типов он дает возможность отметить, что у переменной может потенциально отсутствовать значение. Мы обсудим класс `Optional<T>` подробнее в главе 11.

Еще одна идея — (*структурное*) сопоставление с шаблоном¹, используемое в математике. Например:

$f(0) = 1$
в противном случае $f(n) = n * f(n-1)$

В Java для этого можно написать оператор `if-then-else` или `switch`. Другие языки продемонстрировали, что в случае более сложных типов данных сопоставление с шаблоном позволяет выразить идеи программирования лаконичнее по сравнению с оператором `if-then-else`. Для таких типов данных в качестве альтернативы `if-then-else` можно также использовать полиморфизм и переопределение методов, но дискуссия относительно того, что уместнее, все еще продолжается². Мы полагаем, что и то и другое — полезные инструменты, которые не будут лишними в вашем арсенале. К сожалению, Java 8 не полностью поддерживает сопоставление с шаблоном,

¹ Эту фразу можно толковать двояко. Одно из толкований знакомо нам из математики и функционального программирования, в соответствии с ним функция определяется операторами `case`, а не с помощью конструкции `if-then-else`. Второе толкование относится к фразам типа «найти все файлы вида 'IMG*.JPG' в заданном каталоге» и связано с так называемыми регулярными выражениями.

² Введение в эту дискуссию можно найти в статье «Википедии», посвященной «проблеме выражения» (*expression problem* — термин, придуманный Филом Уэдлером (Phil Wadler)).

хотя в главе 19 мы покажем, как его можно выразить на этом языке. В настоящий момент обсуждается предложение по расширению Java — добавление в будущие версии языка поддержки сопоставления с шаблоном (см. <http://openjdk.java.net/jeps/305>). Тем временем в качестве иллюстрации приведем пример на языке программирования Scala (еще один Java-подобный язык, использующий JVM и послуживший источником вдохновения для некоторых аспектов эволюции Java; см. главу 20). Допустим, вы хотите написать программу для элементарных упрощений дерева, представляющего арифметическое выражение. В Scala можно написать следующий код для декомпозиции объекта типа `Expr`, служащего для представления подобных выражений, с последующим возвратом другого объекта `Expr`:

```
public class MeaningOfThis {
    public final int value = 4;
    public void doIt() {
        int value = 6;
        Runnable r = new Runnable() {
            public final int value = 5;
            public void run(){
                int value = 10;
                System.out.println(this.value);
            }
        };
        r.run();
    }
    public static void main(String...args) {
        MeaningOfThis m = new MeaningOfThis();
        m.doIt(); ← Что выводится в этой строке?
    }
}
```

Синтаксис `expr match` языка Scala соответствует `switch (expr)` в языке Java. Пока не задумывайтесь над этим кодом — мы расскажем о сопоставлении с шаблоном подробнее в главе 19. Сейчас можете считать сопоставление с шаблоном просто расширенным вариантом `switch`, способным в то же время разбивать тип данных на компоненты.

Но почему нужно ограничивать оператор `switch` в Java простыми типами данных и строками? Функциональные языки программирования обычно позволяют использовать `switch` для множества других типов данных, включая возможность сопоставления с шаблоном (в коде на языке Scala это делается с помощью операции `match`). Для прохода по объектам семейства классов (например, различных компонентов автомобиля: колес (`Wheel`), двигателя (`Engine`), шасси (`Chassis`) и т. д.) с применением какой-либо операции к каждому из них в объектно-ориентированной архитектуре часто применяется паттерн проектирования «Посетитель» (`Visitor`). Одно из преимуществ сопоставления с шаблоном — возможность для компилятора выявлять типичные ошибки вида: «Класс `Brakes` — часть семейства классов, используемых для представления компонентов класса `Car`. Вы забыли обработать его явным образом».

Главы 18 и 19 представляют собой полное введение в функциональное программирование и написание программ в функциональном стиле на языке Java 8, включая описание набора имеющихся в его библиотеке функций. Глава 20 развивает эту тему сравнением возможностей Java 8 с возможностями языка Scala — языка, в основе которого также лежит использование JVM и который эволюционировал настолько

быстро, что уже претендует на часть ниши Java в экосистеме языков программирования. Мы разместили этот материал ближе к концу книги, чтобы вам было понятнее, почему были добавлены те или иные новые возможности Java 8 и Java 9.

Возможности Java 8, 9, 10 и 11: с чего начать?

В Java 8, как и в Java 9, были внесены значительные изменения. Но повседневную практику Java-программистов в масштабе мелких фрагментов кода, вероятно, сильнее всего затронут дополнения, внесенные в восьмую версию: идея передачи метода или лямбда-выражения быстро становится жизненно важным знанием в Java. Напротив, усовершенствования Java 9 расширяют возможности описания и использования крупных компонентов, идет ли речь о модульном структурировании системы или импорте набора инструментов для реактивного программирования. Наконец, Java 10 вносит намного меньшие изменения, по сравнению с предыдущими двумя версиями, — они состоят из правил вывода типов для локальных переменных, которые мы вкратце обсудим в главе 21, где также упомянем связанный с ними расширенный синтаксис для аргументов лямбда-выражений, который введен в Java 11.

Java 11 был выпущен в сентябре 2018 года и включает обновленную асинхронную клиентскую библиотеку HTTP (<http://openjdk.java.net/jeps/321>), использующую новые возможности Java 8 и 9 (подробности см. в главах 15–17) — `CompletableFuture` и реактивное программирование.

Резюме

- ❑ Всегда учитывайте идею экосистемы языков программирования и следующее из нее бремя необходимости для языков эволюционировать или увядать. Хотя на текущий момент Java более чем процветающий язык, можно вспомнить и другие процветавшие языки, такие как COBOL, которые не сумели эволюционировать.
- ❑ Основные новшества Java 8 включают новые концепции и функциональность, облегчающую написание лаконичных и эффективных программ.
- ❑ Возможности многоядерных процессоров плохо использовались в Java до версии 8.
- ❑ Функции стали полноправными значениями; методы теперь можно передавать в виде функциональных значений. Обратите также внимание на возможность написания анонимных функций (лямбда-выражений).
- ❑ Концепция потоков данных Java 8 во многом обобщает понятие коллекций, но зачастую позволяет получить более удобочитаемый код и обрабатывать элементы потока параллельно.
- ❑ Программирование с использованием крупномасштабных компонентов, а также эволюция интерфейсов системы исторически плохо поддерживались языком Java. Методы с реализацией по умолчанию появились в Java 8. Они позволяют расширять интерфейс без изменения всех реализующих его классов.
- ❑ В числе других интересных идей из области функционального программирования — обработка `null` и сопоставление с шаблоном.