

## Об авторе

**Герберт Шилдт** является автором многочисленных книг по программированию, пользующихся большим успехом у читателей в течение почти трех десятилетий, а также признанным авторитетом по языку Java. Его книги продаются миллионными тиражами и переведены на многие языки мира. Его перу принадлежит немало книг по Java, в том числе *Java: руководство для начинающих*, *Java: методики программирования Шилдта*, *SWING: руководство для начинающих*, *Искусство программирования на Java*, а также настоящее издание. Он написал немало книг и по другим языкам программирования, включая C, C++ и C#. Интересуясь всеми аспектами вычислительной техники, Герберт уделяет основное внимание языкам программирования и, в частности, компиляторам, интерпретаторам и языкам управления роботами. Он также проявляет большой интерес к стандартизации языков. Герберт окончил Иллинойский университет, получив обе степени — бакалавра и магистра. Подробнее об авторе можно узнать, посетив его веб-сайт по адресу [www.HerbSchildt.com](http://www.HerbSchildt.com).

## О научном редакторе

**Д-р Дэни Ковард** редактировал все издания этой книги. Он ввел понятие сервлетов Java в первую версию платформы Java EE, внедрил веб-службы на платформе Java ME, составил общую стратегию и план разработки версии Java SE 7. Он также заложил основы технологии JavaFX, а совсем недавно разработал самое крупное дополнение к стандарту Java EE 7 и прикладному программному интерфейсу Java WebSocket API. Д-р Ковард обладает необычайно обширными знаниями всех аспектов технологии Java: от программирования на Java и до разработки прикладных программных интерфейсов API вместе с опытными специалистами в данной области, а также многолетним опытом работы в исполнительном комитете Java Community Process. Он также является автором книги *JavaWebSocket Programming* и готовящейся к изданию книги по Java EE. Д-р Ковард окончил Оксфордский университет, получив степени бакалавра, магистра и доктора математических наук.

# Содержание

<b>Предисловие</b>	28
<b>Часть I. Язык Java</b>	31
<b>Глава 1. История и развитие языка Java</b>	33
Происхождение Java	33
Зарождение современного программирования: язык C	34
Следующий этап: язык C++	35
Предпосылки к созданию Java	37
Создание языка Java	37
Связь с языком C#	39
Каким образом язык Java изменил Интернет	40
Апплеты на Java	40
Безопасность	41
Переносимость	41
Чудо Java: байт-код	41
Сервлеты: серверные программы на Java	43
Отличительные особенности Java	43
Простота	44
Объектная ориентированность	44
Надежность	44
Многопоточность	45
Архитектурная нейтральность	45
Интерпретируемость и высокая производительность	46
Распределенность	46
Динамичность	46
Эволюция языка Java	46
Версия Java SE 8	50
Культура нововведений	51
<b>Глава 2. Краткий обзор Java</b>	53
Объектно-ориентированное программирование	53
Две методики	53
Абстракция	54
Три принципа ООП	54

Первый пример простой программы	60
Ввод кода программы	61
Компиляция программы	61
Подробный анализ первого примера программы	62
Второй пример короткой программы	65
Два управляющих оператора	66
Условный оператор <code>if</code>	67
Оператор цикла <code>for</code>	68
Использование блоков кода	69
Вопросы лексики	70
Пробелы	71
Идентификаторы	71
Литералы	71
Комментарии	72
Разделители	72
Ключевые слова Java	72
Библиотеки классов Java	73
<b>Глава 3. Типы данных, переменные и массивы</b>	<b>75</b>
Java – строго типизированный язык	75
Примитивные типы	75
Целые числа	76
Тип <code>byte</code>	77
Тип <code>short</code>	77
Тип <code>int</code>	77
Тип <code>long</code>	78
Числа с плавающей точкой	78
Тип <code>float</code>	79
Тип <code>double</code>	79
Символы	80
Логические значения	81
Подробнее о литералах	82
Целочисленные литералы	82
Литералы с плавающей точкой	84
Логические литералы	85
Символьные литералы	85
Строковые литералы	86
Переменные	86
Объявление переменной	86
Динамическая инициализация	87
Область и срок действия переменных	87
Преобразование и приведение типов	90
Автоматическое преобразование типов в Java	90
Приведение несовместимых типов	91
Автоматическое продвижение типов в выражениях	92
Правила продвижения типов	93
Массивы	94
Одномерные массивы	94
Многомерные массивы	97
Альтернативный синтаксис объявления массивов	101
Введение в символьные строки	101
Замечание по поводу указателей для программирующих на C/C++	102

## 8 Содержание

<b>Глава 4 . Операции</b>	103
Арифметические операции	103
Основные арифметические операции	104
Операция деления по модулю	105
Составные арифметические операции с присваиванием	105
Операции инкремента и декремента	106
Поразрядные операции	108
Поразрядные логические операции	109
Сдвиг влево	111
Сдвиг вправо	113
Беззнаковый сдвиг вправо	114
Поразрядные составные операции с присваиванием	116
Операции отношения	117
Логические операции	118
Укороченные логические операции	120
Операция присваивания	120
Тернарная операция ?	121
Предшествование операций	122
Применение круглых скобок	123
<b>Глава 5. Управляющие операторы</b>	125
Операторы выбора	125
Условный оператор <code>if</code>	125
Оператор <code>switch</code>	128
Операторы цикла	133
Цикл <code>while</code>	133
Цикл <code>do-while</code>	135
Цикл <code>for</code>	138
Вложенные циклы	147
Операторы перехода	147
Применение оператора <code>break</code>	148
Применение оператора <code>continue</code>	151
Оператор <code>return</code>	153
<b>Глава 6. Введение в классы</b>	155
Основы классов	155
Общая форма класса	155
Простой класс	156
Объявление объектов	159
Подробное рассмотрение оператора <code>new</code>	160
Присваивание переменным ссылок на объекты	161
Введение в методы	162
Ввод метода в класс <code>Box</code>	163
Возврат значений	164
Ввод метода, принимающего параметры	166
Конструкторы	168
Параметризованные конструкторы	170
Ключевое слово <code>this</code>	171
Соккрытие переменной экземпляра	172
Сборка “мусора”	172
Метод <code>finalize()</code>	173
Класс <code>Stack</code>	174

<b>Глава 7. Подробное рассмотрение классов и методов</b>	177
Перегрузка методов	177
Перегрузка конструкторов	180
Применение объектов в качестве параметров	182
Подробное рассмотрение особенностей передачи аргументов	184
Возврат объектов	186
Рекурсия	187
Введение в управление доступом	189
Ключевое слово <code>static</code>	193
Ключевое слово <code>final</code>	194
Еще раз о массивах	195
Вложенные и внутренние классы	197
Краткий обзор класса <code>String</code>	200
Применение аргументов командной строки	202
Аргументы переменной длины	203
Перегрузка методов с аргументами переменной длины	206
Аргументы переменной длины и неоднозначность	207
<b>Глава 8. Наследование</b>	209
Основы наследования	209
Доступ к членам класса и наследование	211
Более практический пример	212
Переменная суперкласса может ссылаться на объект подкласса	213
Ключевое слово <code>super</code>	214
Вызов конструкторов суперкласса с помощью ключевого слова <code>super</code>	215
Другое применение ключевого слова <code>super</code>	218
Создание многоуровневой иерархии	219
Порядок вызова конструкторов	222
Переопределение методов	223
Динамическая диспетчеризация методов	225
Назначение переопределенных методов	226
Применение переопределения методов	227
Применение абстрактных классов	228
Ключевое слово <code>final</code> в сочетании с наследованием	231
Предотвращение переопределения с помощью ключевого слова <code>final</code>	231
Предотвращения наследования с помощью ключевого слова <code>final</code>	232
Класс <code>Object</code>	233
<b>Глава 9. Пакеты и интерфейсы</b>	235
Пакеты	235
Определение пакета	236
Поиск пакетов и переменная окружения <code>CLASSPATH</code>	236
Краткий пример пакета	237
Защита доступа	238
Пример защиты доступа	239
Импорт пакетов	242
Интерфейсы	244

## 10 Содержание

Объявление интерфейса	245
Реализация интерфейсов	246
Вложенные интерфейсы	249
Применение интерфейсов	250
Переменные в интерфейсах	253
Расширение интерфейсов	255
Методы по умолчанию	255
Основы применения методов по умолчанию	257
Более практический пример	258
Вопросы множественного наследования	259
Применение статических методов в интерфейсе	260
Заключительные соображения по поводу пакетов и интерфейсов	261
<b>Глава 10. Обработка исключений</b>	<b>263</b>
Основы обработки исключений	263
Типы исключений	264
Необрабатываемые исключения	265
Использование блоков операторов <code>try</code> и <code>catch</code>	266
Вывод описания исключения	268
Применение нескольких операторов <code>catch</code>	268
Вложенные операторы <code>try</code>	270
Оператор <code>throw</code>	272
Оператор <code>throws</code>	273
Оператор <code>finally</code>	275
Встроенные в Java исключения	276
Создание собственных подклассов исключений	278
Цепочки исключений	280
Недавно внедренные средства для обработки исключений	282
Применение исключений	284
<b>Глава 11. Многопоточное программирование</b>	<b>285</b>
Модель потоков исполнения в Java	286
Приоритеты потоков	287
Синхронизация	288
Обмен сообщениями	289
Класс <code>Thread</code> и интерфейс <code>Runnable</code>	289
Главный поток исполнения	290
Создание потока исполнения	291
Реализация интерфейса <code>Runnable</code>	292
Расширение класса <code>Thread</code>	294
Выбор способа создания потоков исполнения	295
Создание многих потоков исполнения	295
Применение методов <code>isAlive()</code> и <code>join()</code>	297
Приоритеты потоков исполнения	299
Синхронизация	300
Применение синхронизированных методов	301
Оператор <code>synchronized</code>	303
Взаимодействие потоков исполнения	304
Взаимная блокировка	309

Приостановка, возобновление и остановка потоков исполнения	311
Получение состояния потока исполнения	313
Применение многопоточности	315
<b>Глава 12. Перечисления, автоупаковка и аннотации (метаданные)</b>	<b>317</b>
Перечисления	317
Основные положения о перечислениях	317
Методы <code>values()</code> и <code>valueOf()</code>	319
Перечисления в Java относятся к типам классов	321
Перечисления наследуются от класса <code>Enum</code>	323
Еще один пример перечисления	325
Оболочки типов	326
Класс <code>Character</code>	327
Класс <code>Boolean</code>	327
Оболочки числовых типов	327
Автоупаковка	328
Автоупаковка и методы	329
Автоупаковка и автораспаковка в выражениях	330
Автоупаковка и распаковка значений из классов <code>Boolean</code> и <code>Character</code>	332
Автоупаковка и автораспаковка помогают предотвратить ошибки	333
Предупреждение	333
Аннотации (метаданные)	334
Основы аннотирования программ	334
Правила удержания аннотаций	335
Получение аннотаций во время выполнения с помощью рефлексии	336
Второй пример применения рефлексии	338
Получение всех аннотаций	339
Интерфейс <code>AnnotatedElement</code>	341
Использование значений по умолчанию	341
Аннотации-маркеры	343
Одночленные аннотации	343
Встроенные аннотации	345
Типовые аннотации	347
Повторяющиеся аннотации	352
Некоторые ограничения на аннотации	353
<b>Глава 13. Ввод-вывод, апплеты и прочие вопросы</b>	<b>355</b>
Основы ввода-вывода	355
Потоки ввода-вывода	356
Потоки ввода-вывода байтов и символов	356
Предопределенные потоки ввода-вывода	359
Чтение данных, вводимых с консоли	359
Чтение символов	360
Чтение символьных строк	361
Запись данных, выводимых на консоль	362
Класс <code>PrintWriter</code>	363
Чтение и запись данных в файлы	364
Автоматическое закрытие файла	370
Основы создания апплетов	374

## 12 Содержание

Модификаторы доступа <code>transient</code> и <code>volatile</code>	377
Применение оператора <code>instanceof</code>	378
Модификатор доступа <code>strictfp</code>	380
Платформенно-ориентированные методы	381
Трудности, связанные с платформенно-ориентированными методами	384
Применение ключевого слова <code>assert</code>	385
Параметры включения и отключения режима проверки утверждений	388
Статический импорт	388
Вызов перегружаемых конструкторов по ссылке <code>this()</code>	391
Компактные профили Java API	393
<b>Глава 14. Обобщения</b>	<b>395</b>
Что такое обобщения	396
Простой пример обобщения	396
Обобщения действуют только со ссылочными типами	400
Обобщенные типы различаются по аргументам типа	400
Каким образом обобщения повышают типовую безопасность	401
Обобщенный класс с двумя параметрами типа	403
Общая форма обобщенного класса	404
Ограниченные типы	405
Применение метасимвольных аргументов	407
Ограниченные метасимвольные аргументы	410
Создание обобщенного метода	414
Обобщенные конструкторы	417
Обобщенные интерфейсы	417
Базовые типы и унаследованный код	420
Иерархии обобщенных классов	422
Применение обобщенного суперкласса	422
Обобщенный подкласс	424
Сравнение типов в обобщенной иерархии во время выполнения	425
Приведение типов	428
Переопределение методов в обобщенном классе	428
Выведение типов и обобщения	429
Стирание	431
Мостовые методы	431
Ошибки неоднозначности	433
Некоторые ограничения, присущие обобщениям	434
Получить экземпляр по параметру типа нельзя	434
Ограничения на статические члены	435
Ограничения на обобщенные массивы	435
Ограничения на обобщенные исключения	436
<b>Глава 15. Лямбда-выражения</b>	<b>437</b>
Введение в лямбда-выражения	438
Основные положения о лямбда-выражениях	438
Функциональные интерфейсы	439
Некоторые примеры лямбда-выражений	440
Блочные лямбда-выражения	444
Обобщенные функциональные интерфейсы	446
Передача лямбда-выражений в качестве аргументов	447
Лямбда-выражения и исключения	450

Лямбда-выражения и захват переменных	451
Ссылки на методы	453
Ссылки на статические методы	453
Ссылки на методы экземпляра	454
Ссылки на обобщенные методы	458
Ссылки на конструкторы	460
Предопределенные функциональные интерфейсы	465
<b>Часть II. Библиотека Java</b>	<b>467</b>
<b>Глава 16. Обработка символьных строк</b>	<b>469</b>
Конструкторы символьных строк	470
Длина символьной строки	472
Специальные строковые операции	472
Строковые литералы	472
Сцепление строк	473
Сцепление символьных строк с другими типами данных	473
Преобразование символьных строк и метод <code>toString()</code>	474
Извлечение символов	475
Метод <code>charAt()</code>	475
Метод <code>getChars()</code>	476
Метод <code>getBytes()</code>	476
Метод <code>toCharArray()</code>	477
Сравнение символьных строк	477
Методы <code>equals()</code> и <code>equalsIgnoreCase()</code>	477
Метод <code>regionMatches()</code>	478
Методы <code>startsWith()</code> и <code>endsWith()</code>	478
Метод <code>equals()</code> в сравнении с операцией <code>==</code>	479
Метод <code>compareTo()</code>	480
Поиск в символьных строках	481
Видоизменение символьных строк	482
Метод <code>substring()</code>	483
Метод <code>concat()</code>	484
Метод <code>replace()</code>	484
Метод <code>trim()</code>	484
Преобразование данных методом <code>valueOf()</code>	485
Изменение регистра символов в строке	486
Соединение символьных строк	487
Дополнительные методы из класса <code>String</code>	487
Класс <code>StringBuffer</code>	489
Конструкторы класса <code>StringBuffer</code>	489
Методы <code>length()</code> и <code>capacity()</code>	490
Метод <code>ensureCapacity()</code>	490
Метод <code>setLength()</code>	491
Методы <code>charAt()</code> и <code>setCharAt()</code>	491
Метод <code>getChars()</code>	492
Метод <code>append()</code>	492
Метод <code>insert()</code>	493
Метод <code>reverse()</code>	493
Методы <code>delete()</code> и <code>deleteCharAt()</code>	494

## 14 Содержание

Метод <code>replace()</code>	494
Метод <code>substring()</code>	495
Дополнительные методы из класса <code>StringBuffer</code>	495
Класс <code>StringBuilder</code>	496

### Глава 17. Пакет `java.lang` 497

Оболочки примитивных типов	498
Класс <code>Number</code>	498
Классы <code>Double</code> и <code>Float</code>	498
Методы <code>isInfinite()</code> и <code>isNaN()</code>	503
Классы <code>Byte</code> , <code>Short</code> , <code>Integer</code> и <code>Long</code>	503
Класс <code>Character</code>	514
Дополнения класса <code>Character</code> для поддержки кодовых точек в Юникоде	517
Класс <code>Boolean</code>	519
Класс <code>Void</code>	520
Класс <code>Process</code>	520
Класс <code>Runtime</code>	521
Управление памятью	523
Выполнение других программ	524
Класс <code>ProcessBuilder</code>	525
Класс <code>System</code>	528
Измерение времени выполнения программы методом <code>currentTimeMills()</code>	530
Применение метода <code>arraycopy()</code>	531
Свойства окружения	532
Класс <code>Object</code>	532
Применение метода <code>clone()</code> и интерфейса <code>Cloneable</code>	533
Класс <code>Class</code>	535
Класс <code>ClassLoader</code>	538
Класс <code>Math</code>	539
Тригонометрические функции	539
Экспоненциальные функции	540
Функции округления	540
Прочие методы из класса <code>Math</code>	542
Класс <code>StrictMath</code>	544
Класс <code>Compiler</code>	545
Классы <code>Thread</code> , <code>ThreadGroup</code> и интерфейс <code>Runnable</code>	545
Интерфейс <code>Runnable</code>	545
Класс <code>Thread</code>	545
Класс <code>ThreadGroup</code>	548
Классы <code>ThreadLocal</code> и <code>InheritableThreadLocal</code>	553
Класс <code>Package</code>	553
Класс <code>RuntimePermission</code>	555
Класс <code>Throwable</code>	555
Класс <code>SecurityManager</code>	555
Класс <code>StackTraceElement</code>	555
Класс <code>Enum</code>	556
Класс <code>ClassValue</code>	557
Интерфейс <code>CharSequence</code>	558
Интерфейс <code>Comparable</code>	558
Интерфейс <code>Appendable</code>	559

Интерфейс Iterable	559
Интерфейс Readable	560
Интерфейс AutoCloseable	560
Интерфейс Thread.UncaughtExceptionHandler	560
Подпакеты из пакета в java.lang	561
Пакет java.lang.annotation	561
Пакет java.lang.instrument	561
Пакет java.lang.invoke	561
Пакет java.lang.management	561
Пакет java.lang.ref	562
Пакет java.lang.reflect	562
<b>Глава 18. Пакет java.util, часть I. Collections Framework</b>	<b>563</b>
Краткий обзор коллекций	564
Изменения каркаса коллекций в версии JDK 5	566
Обобщения коренным образом изменили каркас коллекций	566
В средствах автоматической упаковки используются	
примитивные типы данных	567
Цикл for в стиле for each	567
Интерфейсы коллекций	567
Интерфейс Collection	568
Интерфейс List	571
Интерфейс Set	573
Интерфейс SortedSet	573
Интерфейс NavigableSet	574
Интерфейс Queue	576
Интерфейс Dequeue	577
Классы коллекций	579
Класс ArrayList	580
Класс LinkedList	584
Класс HashSet	585
Класс LinkedHashSet	587
Класс TreeSet	587
Класс PriorityQueue	588
Класс ArrayDeque	589
Класс EnumSet	590
Доступ к коллекциям через итератор	591
Применение интерфейса Iterator	593
Цикл for в стиле for each как альтернатива итераторам	595
Итераторы-разделители	596
Сохранение объектов пользовательских классов в коллекциях	599
Интерфейс RandomAccess	600
Обращение с отображениями	601
Интерфейсы отображений	601
Классы отображений	609
Компараторы	614
Применение компараторов	617
Алгоритмы коллекций	622
Массивы	629
Унаследованные классы и интерфейсы	634

## 16 Содержание

Интерфейс Enumeration	635
Класс Vector	635
Класс Stack	639
Класс Dictionary	641
Класс Properties	645
Применение методов store () и load ()	649
Заключительные соображения по поводу коллекций	650
<b>Глава 19. Пакет java.util, часть II. Прочие служебные классы</b>	<b>651</b>
Класс StringTokenizer	651
Класс BitSet	653
Классы Optional, OptionalDouble, OptionalInt и OptionalLong	656
Класс Date	659
Класс Calendar	661
Класс GregorianCalendar	664
Класс TimeZone	666
Класс SimpleTimeZone	667
Класс Locale	668
Класс Random	670
Класс Observable	672
Интерфейс Observer	673
Пример наблюдения за объектами	674
Классы Timer и TimerTask	676
Класс Currency	678
Класс Formatter	680
Конструкторы класса Formatter	680
Методы из класса Formatter	681
Основы форматирования	682
Форматирование строк и символов	684
Форматирование чисел	684
Форматирование времени и даты	685
Спецификаторы формата %n и %%	687
Указание минимальной ширины поля	687
Указание точности	689
Применение признаков формата	689
Выравнивание выводимых данных	690
Признаки пробела, +, 0 и (	691
Признак запятой	692
Признак #	692
Прописные формы спецификаторов формата	692
Применение индекса аргумента	693
Закрытие объекта типа Formatter	694
Аналог функции printf () в Java	695
Класс Scanner	695
Конструкторы класса Scanner	695
Некоторые примеры применения класса Scanner	700
Установка разделителей	704
Прочие средства класса Scanner	705
Классы ResourceBundle, ListResourceBundle и PropertyResourceBundle	706

Прочие служебные классы и интерфейсы	710
Пакеты <code>java.util.concurrent</code> , <code>java.util.concurrent.atomic</code> , <code>java.util.concurrent.locks</code>	712
Пакет <code>java.util.function</code>	712
Пакет <code>java.util.jar</code>	716
Пакет <code>java.util.logging</code>	716
Пакет <code>java.util.prefs</code>	716
Пакет <code>java.util.regex</code>	716
Пакет <code>java.util.spi</code>	716
Пакет <code>java.util.stream</code>	716
Пакет <code>java.util.zip</code>	716
<b>Глава 20. Пакет <code>java.io</code> для ввода-вывода</b>	717
Классы и интерфейсы ввода-вывода	718
Класс <code>File</code>	718
Каталоги	722
Применение интерфейса <code>FilenameFilter</code>	723
Альтернативный метод <code>listFiles()</code>	724
Создание каталогов	724
Интерфейсы <code>AutoCloseable</code> , <code>Closeable</code> и <code>Flushable</code>	724
Исключения ввода-вывода	725
Два способа закрытия потока ввода-вывода	726
Классы потоков ввода-вывода	727
Потоки ввода-вывода байтов	728
Класс <code>InputStream</code>	728
Класс <code>OutputStream</code>	729
Класс <code>FileInputStream</code>	730
Класс <code>FileOutputStream</code>	732
Класс <code>ByteArrayInputStream</code>	734
Класс <code>ByteArrayOutputStream</code>	736
Фильтруемые потоки ввода-вывода байтов	737
Буферизованные потоки ввода-вывода байтов	737
Потоки ввода-вывода символов	748
Класс <code>Reader</code>	749
Класс <code>Writer</code>	750
Класс <code>FileReader</code>	751
Класс <code>FileWriter</code>	751
Класс <code>CharArrayReader</code>	752
Класс <code>CharArrayWriter</code>	754
Класс <code>BufferedReader</code>	755
Класс <code>BufferedWriter</code>	756
Класс <code>PushbackReader</code>	757
Класс <code>PrintWriter</code>	758
Класс <code>Console</code>	759
Сериализация	761
Интерфейс <code>Serializable</code>	762
Интерфейс <code>Externalizable</code>	762
Интерфейс <code>ObjectOutput</code>	763
Класс <code>ObjectOutputStream</code>	763
Интерфейс <code>ObjectInput</code>	764

## 18 Содержание

Класс <code>ObjectInputStream</code>	765
Пример сериализации	767
Преимущества потоков ввода-вывода	768
<b>Глава 21. Система ввода-вывода NIO</b>	<b>769</b>
Классы системы ввода-вывода NIO	769
Основные положения о системе ввода-вывода NIO	770
Буфера	770
Каналы	772
Наборы символов и селекторы	774
Усовершенствования в системе NIO, начиная с версии JDK 7	774
Интерфейс <code>Path</code>	774
Класс <code>Files</code>	776
Класс <code>Paths</code>	779
Интерфейсы атрибутов файлов	780
Классы <code>FileSystem</code> , <code>FileSystems</code> и <code>FileStore</code>	782
Применение системы ввода-вывода NIO	782
Применение системы NIO для канального ввода-вывода	783
Применение системы NIO для потокового ввода-вывода	793
Применение системы ввода-вывода NIO для операций в файловой системе	796
Примеры организации канального ввода-вывода до версии JDK 7	804
Чтение из файла до версии JDK 7	804
Запись в файл версии до JDK 7	807
<b>Глава 22. Работа в сети</b>	<b>811</b>
Основы работы в сети	811
Сетевые классы и интерфейсы	813
Класс <code>InetAddress</code>	813
Фабричные методы	814
Методы экземпляра	815
Классы <code>Inet4Address</code> и <code>Inet6Address</code>	815
Клиентские сокеты по протоколу TCP/IP	816
Класс <code>URL</code>	819
Класс <code>URLConnection</code>	821
Класс <code>HttpURLConnection</code>	824
Класс <code>URI</code>	826
Cookie-файлы	826
Серверные сокеты по протоколу TCP/IP	827
Дейтаграммы	827
Класс <code>DatagramSocket</code>	828
Класс <code>DatagramPacket</code>	829
Пример обработки дейтаграмм	830
<b>Глава 23. Класс <code>Applet</code></b>	<b>833</b>
Два типа апплетов	833
Основы разработки апплетов	834
Класс <code>Applet</code>	835
Структура апплетов	837
Скелет апплета	838
Инициализация и прекращение работы апплета	839

Переопределение метода update ()	840
Простые методы воспроизведения апплетов	841
Запрос на повторное воспроизведение	843
Простой апплет с баннером	844
Применение строки состояния	846
HTML-дескриптор APPLET	847
Передача параметров апплетам	848
Усовершенствование апплета, воспроизводящего баннер	850
Методы getDocumentBase () и getCodeBase ()	851
Интерфейс AppletContext и метод showDocument ()	852
Интерфейс AudioClip	854
Интерфейс AppletStub	854
Консольный вывод	854
<b>Глава 24. Обработка событий</b>	<b>855</b>
Два механизма обработки событий	855
Модель делегирования событий	856
События	856
Источники событий	857
Приемники событий	857
Классы событий	858
Класс ActionEvent	859
Класс AdjustmentEvent	860
Класс ComponentEvent	861
Класс ContainerEvent	861
Класс FocusEvent	862
Класс InputEvent	863
Класс ItemEvent	864
Класс KeyEvent	864
Класс MouseEvent	865
Класс MouseWheelEvent	867
Класс TextEvent	868
Класс WindowEvent	868
Источники событий	870
Интерфейсы приемников событий	870
Интерфейс ActionListener	871
Интерфейс AdjustmentListener	871
Интерфейс ComponentListener	872
Интерфейс ContainerListener	872
Интерфейс FocusListener	872
Интерфейс ItemListener	872
Интерфейс KeyListener	872
Интерфейс MouseListener	873
Интерфейс MouseMotionListener	873
Интерфейс MouseWheelListener	873
Интерфейс TextListener	873
Интерфейс WindowFocusListener	873
Интерфейс WindowListener	873
Применение модели делегирования событий	874
Обработка событий от мыши	874
Обработка событий от клавиатуры	877

Классы адаптеров	880
Внутренние классы	882
Анонимные внутренние классы	883
<b>Глава 25. Введение в библиотеку AWT: работа с окнами, графикой и текстом</b>	<b>885</b>
Классы библиотеки AWT	886
Основные положения об окнах	888
Класс Component	889
Класс Container	889
Класс Panel	889
Класс Window	890
Класс Frame	890
Класс Canvas	890
Работа с обрамляющими окнами	890
Установка размеров окна	891
Соккрытие и отображение окна	891
Установка заголовка окна	891
Заккрытие обрамляющего окна	891
Создание обрамляющего окна в апплете, построенном на основе библиотеки AWT	892
Обработка событий в обрамляющем окне	894
Создание оконной прикладной программы	898
Отображение информации в окне	899
Поддержка графики	900
Рисование линий	900
Рисование прямоугольников	900
Рисование эллипсов и окружностей	901
Рисование дуг	901
Рисование многоугольников	901
Демонстрация методов рисования	902
Изменение размеров графики	902
Работа с цветом	904
Методы из класса Color	905
Установка режима рисования	907
Работа со шрифтами	909
Определение доступных шрифтов	910
Создание и выбор шрифта	911
Получение сведений о шрифте	913
Управление форматированием выводимого текста	914
Отображение многострочного текста	915
Центровка текста	917
Выравнивание многострочного текста	918
<b>Глава 26. Применение элементов управления, диспетчеров компоновки и меню из библиотеки AWT</b>	<b>923</b>
Основные положения об элементах управления	924
Ввод и удаление элементов управления	924
Реагирование на элементы управления	925
Исключение типа HeadlessException	925

Метки	925
Экранные кнопки	926
Обработка событий от кнопок	927
Флажки	930
Обработка событий от флажков	931
Кнопки-переключатели	932
Элементы управления выбором	934
Обработка событий от раскрывающихся списков	935
Использование списков	936
Обработка событий от списков	938
Управление полосами прокрутки	939
Обработка событий от полос прокрутки	941
Текстовые поля	943
Обработка событий в текстовых полях	944
Текстовые области	945
Диспетчеры компоновки	947
Класс <code>FlowLayout</code>	948
Класс <code>BorderLayout</code>	950
Вставки	952
Класс <code>GridLayout</code>	953
Класс <code>CardLayout</code>	954
Класс <code>GridBagLayout</code>	957
Меню и строки меню	962
Диалоговые окна	968
Диалоговые окна выбора файлов	972
О переопределении метода <code>paint()</code>	973
<b>Глава 27. Изображения</b>	<b>975</b>
Форматы файлов	975
Основы работы с изображениями: создание, загрузка и отображение	976
Создание объекта класса <code>Image</code>	976
Загрузка изображения	977
Воспроизведение изображения	977
Интерфейс <code>ImageObserver</code>	979
Двойная буферизация	980
Класс <code>MediaTracker</code>	983
Интерфейс <code>ImageProducer</code>	986
Класс <code>MemoryImageSource</code>	986
Интерфейс <code>ImageConsumer</code>	988
Класс <code>PixelGrabber</code>	988
Класс <code>ImageFilter</code>	991
Класс <code>CropImageFilter</code>	991
Фильтр класса <code>RGBImageFilter</code>	992
Дополнительные классы для формирования изображений	1004
<b>Глава 28. Утилиты параллелизма</b>	<b>1005</b>
Пакеты параллельного API	1006
Пакет <code>java.util.concurrent.atomic</code>	1008
Пакет <code>java.util.concurrent.locks</code>	1008
Применение объектов синхронизации	1008

## 22 Содержание

Класс Semaphore	1008
Класс CountdownLatch	1014
Класс CyclicBarrier	1016
Класс Exchanger	1018
Класс Phaser	1020
Применение исполнителя	1028
Простой пример исполнителя	1029
Применение интерфейсов Callable и Future	1031
Перечисление TimeUnit	1034
Параллельные коллекции	1035
Блокировки	1036
Атомарные операции	1039
Параллельное программирование средствами Fork/Join Framework	1040
Основные классы Fork/Join Framework	1041
Стратегия “разделяй и властвуй”	1045
Первый простой пример вилочного соединения	1047
Влияние уровня параллелизма	1049
Пример применения класса RecursiveTask<V>	1053
Асинхронное выполнение задач	1055
Отмена задачи	1056
Определение состояния завершения задачи	1056
Перезапуск задачи	1056
Предмет дальнейшего изучения	1057
Рекомендации относительно вилочного соединения	1059
Утилиты параллелизма в сравнении с традиционным подходом к многозадачности в Java	1059
<b>Глава 29. Поточковый API</b>	<b>1061</b>
Основные положения о потоках данных	1061
Потоковые интерфейсы	1062
Получение потока данных	1065
Простой пример потока данных	1066
Операции сведения	1070
Параллельные потоки данных	1073
Отображение	1075
Накопление	1079
Итераторы и потоки данных	1083
Применение итератора в потоке данных	1083
Применение итератора-разделителя	1085
Дальнейшее изучение потокового API	1087
<b>Глава 30. Регулярные выражения и другие пакеты</b>	<b>1089</b>
Пакеты из базового API	1089
Обработка регулярных выражений	1092
Класс Pattern	1092
Класс Matcher	1092
Синтаксис регулярных выражений	1093
Примеры, демонстрирующие совпадение с шаблоном	1094
Два варианта сопоставления с шаблоном	1100
Дальнейшее изучение регулярных выражений	1100
Рефлексия	1101

Удаленный вызов методов	1105
Простое приложение “клиент–сервер”, использующее механизм RMI	1105
Форматирование даты и времени средствами пакета <code>java.text</code>	1109
Класс <code>DateFormat</code>	1109
Класс <code>SimpleDateFormat</code>	1111
API даты и времени, внедренный в версии JDK 8	1113
Основные классы даты и времени	1113
Форматирование даты и времени	1115
Синтаксический анализ символьных строк даты и времени	1117
Дальнейшее изучение пакета <code>java.time</code>	1118
<b>Часть III. Введение в программирование ГПИ средствами Swing</b>	1119
<b>Глава 31. Введение в библиотеку Swing</b>	1121
Происхождение библиотеки Swing	1121
Построение библиотеки Swing на основе библиотеки AWT	1122
Главные особенности библиотеки Swing	1122
Легковесные компоненты Swing	1123
Подключаемый стиль оформления	1123
Связь с архитектурой MVC	1124
Компоненты и контейнеры	1125
Компоненты	1125
Контейнеры	1126
Панели контейнеров верхнего уровня	1126
Пакеты библиотеки Swing	1127
Простое Swing-приложение	1127
Обработка событий	1132
Создание Swing-апплета	1135
Рисование средствами Swing	1138
Основы рисования	1138
Вычисление области рисования	1139
Пример рисования	1140
<b>Глава 32. Исследование библиотеки Swing</b>	1143
Классы <code>JLabel</code> и <code>ImageIcon</code>	1143
Класс <code>JTextField</code>	1145
Кнопки из библиотеки Swing	1147
Класс <code>JButton</code>	1148
Класс <code>JToggleButton</code>	1150
Флажки	1153
Кнопки-переключатели	1155
Класс <code>JTabbedPane</code>	1157
Класс <code>JScrollPane</code>	1160
Класс <code>JList</code>	1162
Класс <code>JComboBox</code>	1166
Деревья	1168
Класс <code>JTable</code>	1172
Основные положения о меню	1175

## 24 Содержание

<b>Глава 33. Введение в меню Swing</b>	1175
Краткий обзор классов JMenuBar, JMenu и JMenuItem	1177
Класс JMenuBar	1177
Класс JMenu	1178
Класс JMenuItem	1180
Создание главного меню	1180
Ввод мнемоники и оперативных клавиш в меню	1184
Ввод изображений и всплывающих подсказок в пункты меню	1187
Классы JRadioButtonMenuItem и JCheckBoxMenuItem	1188
Создание всплывающего меню	1190
Создание панели инструментов	1193
Действия	1196
Составление окончательного варианта программы MenuDemo	1202
Дальнейшее изучение библиотеки Swing	1208
<b>Часть IV. Введение в программирование ГПИ средствами JavaFX</b>	1209
<b>Глава 34. Введение в JavaFX</b>	1211
Основные понятия JavaFX	1212
Пакеты JavaFX	1212
Классы подмостков и сцены	1213
Узлы и графы сцены	1213
Компоновки	1213
Класс приложения и методы его жизненного цикла	1214
Запуск JavaFX-приложения	1214
Скелет JavaFX-приложения	1215
Компиляция и выполнение JavaFX-приложения	1218
Поток исполнения приложения	1219
Метка – простейший элемент управления в JavaFX	1219
Применение кнопок и событий	1221
Основа обработки событий в JavaFX	1222
Элемент управления экранной кнопкой	1223
Демонстрация обработки событий на примере экранных кнопок	1224
Рисование непосредственно на холсте	1227
<b>Глава 35. Элементы управления JavaFX</b>	1233
Классы Image и ImageView	1233
Ввод изображения в метку	1236
Применение изображения в экранной кнопке	1238
Класс ToggleButton	1240
Класс RadioButton	1243
Обработка событий изменения в группе кнопок-переключателей	1246
Другой способ управления кнопками-переключателями	1248
Класс CheckBox	1251
Класс ListView	1254
Представление списка с полосами прокрутки	1258
Активизация режима одновременного выбора нескольких элементов из списка	1258
Класс ComboBox	1260
Класс TextField	1263
Класс ScrollPane	1265
Класс TreeView	1269

Эффекты и преобразования	1273
Эффекты	1274
Преобразования	1275
Демонстрация эффектов и преобразований	1276
Ввод всплывающих подсказок	1279
Отключение элементов управления	1280
<b>Глава 36. Введение в меню JavaFX</b>	<b>1281</b>
Основные положения о меню	1281
Краткий обзор классов MenuBar, Menu и MenuItem	1283
Класс MenuBar	1283
Класс Menu	1284
Класс MenuItem	1285
Создание главного меню	1286
Ввод мнемоники и оперативных клавиш в меню	1291
Ввод изображений в пункты меню	1293
Классы RadioMenuItem и CheckMenuItem	1294
Создание контекстного меню	1296
Создание панели инструментов	1300
Составление окончательного варианта приложения MenuDemo	1302
Дальнейшее изучение JavaFX	1308
<b>Часть V. Применение Java</b>	<b>1309</b>
<b>Глава 37. Компоненты Java Beans</b>	<b>1311</b>
Общее представление о компонентах Java Beans	1311
Преимущества компонентов Java Beans	1312
Самоанализ	1312
Шаблоны проектирования для свойств компонентов Java Beans	1313
Шаблоны проектирования для событий	1314
Методы и шаблоны проектирования	1315
Применение интерфейса BeanInfo	1315
Привязанные и ограниченные свойства	1315
Сохраняемость компонентов Java Beans	1316
Настройщики	1316
Прикладной программный интерфейс Java Beans API	1317
Класс Introspector	1319
КлассPropertyDescriptor	1319
КлассEventSetDescriptor	1319
КлассMethodDescriptor	1319
Пример компонента Java Bean	1320
<b>Глава 38. Введение в сервлеты</b>	<b>1323</b>
Предпосылки для разработки сервлетов	1323
Жизненный цикл сервлета	1324
Варианты разработки сервлетов	1325
Применение контейнера сервлетов Tomcat	1325
Простой пример сервлета	1327
Создание и компиляция исходного кода сервлета	1327
Запуск контейнера сервлетов Tomcat на выполнение	1328
Запуск веб-браузера и запрос сервлета	1328

Прикладной программный интерфейс Servlet API	1329
Пакет javax.servlet	1329
Интерфейс Servlet	1330
Интерфейс ServletConfig	1330
Интерфейс ServletContext	1331
Интерфейс ServletRequest	1331
Интерфейс ServletResponse	1332
Класс GenericServlet	1333
Класс ServletInputStream	1333
Класс ServletOutputStream	1333
Класс ServletException	1333
Ввод параметров сервлета	1334
Пакет javax.servlet.http	1335
Интерфейс HttpServletRequest	1336
Интерфейс HttpServletResponse	1337
Интерфейс HttpSession	1338
Класс Cookie	1339
Класс HttpServlet	1340
Обработка HTTP-запросов и ответов HTTP	1341
Обработка HTTP-запросов типа GET	1341
Обработка HTTP-запросов типа POST	1342
Применение cookie-файлов	1343
Отслеживание сеансов связи	1346
<b>Приложение А. Применение документирующих комментариев в Java</b>	<b>1347</b>
Дескрипторы утилиты javadoc	1347
Дескриптор \$author	1348
Дескриптор {@code}	1349
Дескриптор @deprecated	1349
Дескриптор {@docRoot}	1349
Дескриптор @exception	1349
Дескриптор {@inheritDoc}	1349
Дескриптор {@link}	1350
Дескриптор {@linkplain}	1350
Дескриптор {@literal}	1350
Дескриптор @param	1350
Дескриптор @return	1350
Дескриптор @see	1350
Дескриптор @serial	1351
Дескриптор @serialData	1351
Дескриптор @serialField	1351
Дескриптор @since	1351
Дескриптор @throws	1352
Дескриптор {@value}	1352
Дескриптор @version	1352
Общая форма документирующих комментариев	1352
Результаты, выводимые утилитой javadoc	1353
Пример применения документирующих комментариев	1353
<b>Предметный указатель</b>	<b>1355</b>

# Предисловие

Java — один из самых важных и широко применяемых языков программирования в мире на протяжении многих лет. В отличие от некоторых других языков программирования, влияние Java не только не уменьшилось со временем, а, наоборот, возросло. С момента первого выпуска он выдвинулся на передний край программирования приложений для Интернета. И каждая последующая версия лишь укрепляла эту позицию. Ныне Java по-прежнему остается первым и самым лучшим языком для разработки веб-ориентированных приложений. Проще говоря, *большая часть современного кода написана на Java*. И это свидетельствует об особом значении языка Java для программирования.

Основная причина успеха Java — его гибкость. Начиная с первой версии 1.0, этот язык непрерывно адаптируется к изменениям в среде программирования и подходам к написанию программ. А самое главное — он не просто следует тенденциям в программировании, а *помогает их создавать*. Способность Java адаптироваться к быстрым изменениям в вычислительной технике служит основной причиной, по которой этот язык программирования продолжает оставаться столь успешным.

Со времени публикации первого издания этой книги в 1996 году она претерпела немало изменений, которые отражали последовательное развитие языка Java. Настоящее, девятое, издание обновлено по версии Java SE 8 (JDK 8). А это означает, что оно содержит немало нового материала, поскольку в версии Java SE 8 появился ряд новых языковых средств. Наиболее важными из них являются лямбда-выражения, вводящие совершенно новый синтаксис и существенно повышающие выразительную силу языка. Вследствие особого значения лямбда-выражений им посвящена отдельная глава, а примеры их применения приведены и в других главах. Внедрение лямбда-выражений повлекло за собой появление других новых языковых средств. К их числу относится библиотека потоков ввода-вывода, входящая в пакет `java.util.stream` и поддерживающая конвейерные операции с данными. Этой библиотеке также посвящена отдельная глава. Еще одним нововведением является метод по умолчанию, позволяющий вводить в интерфейс функциональные возможности, используемые по умолчанию. А такие средства, как повторяющиеся и типовые аннотации, дополнительно повышают эффективность Java. В версии Java SE 8 сделаны также значительные усовершенствования в библиотеке Java API, и некоторые из них описываются в этой книге.

Еще одним важным дополнением настоящего издания служит описание JavaFX — новой технологии Java для построения графического пользовательского интерфейса (ГПИ) приложений на Java. В перспективе JavaFX отводится существенная роль в разработке приложений на Java, и поэтому этой технологии посвящены три отдельные главы. Это означает, что программирующим на Java просто необходимо овладеть технологией JavaFX. Еще одна глава книги посвящена разработке меню средствами Swing. Несмотря на то что технология JavaFX может в перспективе полностью заменить технологию Swing, на момент выхода настоящего издания Swing по-прежнему широко применяется для построения ГПИ приложений на Java, и поэтому подробное рассмотрение этой технологии в настоящем издании вполне обоснованно. И наконец, в отдельных главах упоминаются многочисленные мелкие усовершенствования в версии Java SE 8.

## Книга для всех программистов

Эта книга предназначена для всех категорий программистов: от начинающих до опытных. Начинающий программист найдет в ней подробные пошаговые описания и немало полезных примеров написания кода на Java, а углубленное рассмотрение более сложных функций и библиотек Java должно привлечь внимание опытных программистов. Для обеих категорий читателей в книге указаны действующие ресурсы и полезные ссылки.

## Структура книги

Эта книга служит исчерпывающим справочным пособием по языку Java, в котором описываются его синтаксис, ключевые слова и основополагающие принципы программирования. В ней рассматривается также значительная часть библиотеки Java API. Книга разделена на пять частей, каждая из которых посвящена отдельному аспекту среды программирования Java.

В части I представлено подробное учебное пособие по языку Java. Она начинается с рассмотрения таких основных понятий, как типы данных, операции, управляющие операторы и классы. Затем описываются правила наследования, пакеты, интерфейсы, обработка исключений и многопоточная обработка. Далее рассматриваются аннотации, перечисления, автоупаковка, обобщения, операции ввода-вывода и апплеты. А заключительная глава этой части посвящена лямбда-выражениям, которые, как упоминалось ранее, являются самым важным нововведением в версии Java SE 8.

В части II описываются основные компоненты стандартной библиотеки Java API. В ней обсуждаются следующие вопросы: символьные строки, операции ввода-вывода, работа в сети, стандартные утилиты, каркас коллекций Collections Framework, апплеты, библиотека AWT, обработка событий, формирование изображений, параллельная обработка (включая каркас Fork/Join Framework), регулярные выражения и новая библиотека потоков ввода-вывода.

Часть III состоит из трех глав, посвященных технологии Swing, а часть IV — из такого же количества глав, посвященных технологии JavaFX.

Часть V состоит из двух глав с примерами практического применения Java. Сначала в ней рассматривается технология Java Beans, а затем представлены сервлеты.

## Исходный код примеров, доступный в Интернете

Помните, что исходный код всех примеров, приведенных в этой книге, доступен на веб-сайте издательства Oracle Press по адресу <http://www.mhprofessional.com/product.php?isbn=0071808558> или с сайта Издательского дома “Вильямс” по адресу: <http://archive.williamspublishing.com/cgi-bin/materials.cgi?isbn=978-5-8459-1918-2>.

## Особые благодарности

Выражаю особую благодарность Патрику Нотону (Patrick Naughton), Джо О’Нилу (Joe O’Neil) и Дэнни Коварду (Danny Coward).

Патрик Нотон был одним из создателей языка Java. Он помог мне в написании первого издания этой книги. Значительная часть материала глав 20, 22 и 27 была предоставлена Патриком. Его проницательность, опыт и энергия в огромной степени способствовали успеху этой книги.

При подготовке второго и третьего изданий этой книги Джо О’Нил предоставил исходные черновые материалы, которые послужили основанием для написания глав 30, 32, 37 и 38. Джо помогал мне при написании нескольких книг, и я высоко ценю его вклад.

Дэнни Ковард выполнил научное редактирование книги. Он принимал участие в работе над несколькими моими книгами, и его полезные советы, поучительные замечания и дельные предложения всегда были ценными и достойными большой признательности.

*Герберт Шилдт*

## Дополнительная литература

Настоящее издание открывает серию книг по программированию на Java, написанных Гербертом Шилдтом. Ниже перечислены другие книги этого автора, которые могут вас заинтересовать.

- *Java: методики программирования Шилдта*. И.Д. “Вильямс”, 2008 г.
- *Java: руководство для начинающих*, 6-е изд. И.Д. “Вильямс”, 2015 г.
- *SWING: руководство для начинающих*. И.Д. “Вильямс”, 2007 г.
- *Искусство программирования на Java*. И.Д. “Вильямс”, 2005 г.

Начиная с версии 1.4 в Java предоставляется вторая система ввода-вывода под названием NIO (сокращение от New I/O — новый ввод-вывод). В этой системе поддерживается канальный подход к операциям ввода-вывода, ориентированный на применение буферов. А в версии JDK 7 система ввода-вывода NIO была существенно расширена, и теперь она оказывает улучшенную поддержку средств обработки файлов и файловых систем. На самом деле изменения в этой системе настолько значительны, что она нередко обозначается термином *NIO.2*. Благодаря возможностям, предоставляемым новыми классами файлов из системы ввода-вывода NIO, ожидается, что значение этой системы в обработке файлов будет только возрастать. В этой главе рассматривается ряд основных характеристик системы ввода-вывода NIO.

## Классы системы ввода-вывода NIO

В табл. 21.1 перечислены пакеты, в которых содержатся классы системы ввода-вывода NIO.

**Таблица 21.1. Пакеты, содержащие классы системы ввода-вывода NIO**

Пакет	Назначение
<code>java.nio</code>	Это пакет верхнего уровня в системе ввода-вывода NIO. Он инкапсулирует различные типы буферов, содержащих данные, которыми оперирует система ввода-вывода NIO
<code>java.nio.channels</code>	Поддерживает каналы, открывающие соединения для ввода-вывода
<code>java.nio.channels.spi</code>	Поддерживает поставщики услуг для каналов
<code>java.nio.charset</code>	Инкапсулирует наборы символов. Поддерживает также функционирование кодеров и декодеров для взаимного преобразования символов и байтов
<code>java.nio.charset.spi</code>	Поддерживает поставщики услуг для наборов символов
<code>java.nio.file</code>	Поддерживает ввод-вывод в файлы
<code>java.nio.file.attribute</code>	Поддерживает атрибуты файлов
<code>java.nio.file.spi</code>	Поддерживает поставщики услуг для файловых систем

Прежде чем приступить к рассмотрению системы ввода-вывода NIO, следует заметить, что эта система не предназначена для замены классов ввода-вывода, входящих в состав пакета `java.io` и представленных в главе 20. Напротив, практические знания о классах из этого пакета помогают легче усвоить систему ввода-вывода NIO.

**На заметку!** В этой главе предполагается, что вы уже проработали материал глав 13 и 20, посвященный принципам ввода-вывода вообще и потокового ввода-вывода в частности.

## Основные положения о системе ввода-вывода NIO

Система ввода-вывода NIO построена на двух основополагающих элементах: буферах и каналах. В *буфере* хранятся данные, а *канал* предоставляет открытое соединение с устройством ввода-вывода, например файлом или сокетом. В общем, для применения системы ввода-вывода NIO требуется получить канал для устройства ввода-вывода и буфер для хранения данных. После этого можно обращаться с буфером, вводя или выводя данные по мере необходимости. Поэтому в последующих разделах буфера и каналы будут рассмотрены подробно.

### Буфера

Буфера определяются в пакете `java.nio`. Все буфера являются подклассами, производными от класса `Buffer`, в котором определяются основные функциональные возможности, характерные для каждого буфера, в том числе текущая позиция, предел и емкость. *Текущая позиция* определяет индекс в буфере, с которого в следующий раз начнется операция чтения или записи данных. Текущая позиция перемещается после выполнения большинства операций чтения или записи. *Предел* определяет значение индекса за позицией последней доступной ячейки в буфере. *Емкость* определяет количество элементов, которые можно хранить в буфере. Зачастую предел равен емкости буфера. В классе `Buffer` поддерживается также отметка и очистка буфера. В нем определяется ряд методов, перечисленных в табл. 21.2.

**Таблица 21.2. Методы из класса `Buffer`**

Метод	Описание
<code>abstract Object array()</code>	Возвращает ссылку на массив, если вызывающий буфер поддерживается массивом, иначе генерирует исключение типа <code>UnsupportedOperationException</code> . Если же массив доступен только для чтения, то генерируется исключение типа <code>ReadOnlyBufferException</code>
<code>abstract int arrayOffset()</code>	Возвращает индекс первого элемента массива, если вызывающий буфер поддерживается массивом, а иначе генерируется исключение типа <code>UnsupportedOperationException</code> . Если же массив доступен только для чтения, то генерируется исключение типа <code>ReadOnlyBufferException</code>

Окончание табл. 21.2

Метод	Описание
<code>final int capacity()</code>	Возвращает количество элементов, которые можно хранить в вызывающем буфере
<code>final Buffer clear()</code>	Очищает вызывающий буфер и возвращает ссылку на него
<code>final Buffer flip()</code>	Задает текущую позицию в качестве предела для вызывающего буфера и затем устанавливает текущую позицию в нуль. Возвращает ссылку на буфер
<code>abstract boolean hasArray()</code>	Возвращает логическое значение <b>true</b> , если вызывающий буфер поддерживается массивом, доступным для чтения и записи, а иначе — логическое значение <b>false</b>
<code>final boolean hasRemaining()</code>	Возвращает логическое значение <b>true</b> , если в вызывающем буфере еще остались какие-нибудь элементы, а иначе — логическое значение <b>false</b>
<code>abstract boolean isDirect()</code>	Возвращает логическое значение <b>true</b> , если вызывающий буфер оказывается прямым. Иными словами, операции ввода-вывода выполняются над ним напрямую. В противном случае возвращается логическое значение <b>false</b>
<code>abstract boolean isReadOnly()</code>	Возвращает логическое значение <b>true</b> , если вызывающий буфер является буфером только для чтения, а иначе — логическое значение <b>false</b>
<code>final int limit()</code>	Возвращает предел для вызывающего буфера
<code>final Buffer limit(int n)</code>	Задает предел <i>n</i> для вызывающего буфера. Возвращает ссылку на буфер
<code>final Buffer mark()</code>	Устанавливает метку и возвращает ссылку на вызывающий буфер
<code>final int position()</code>	Возвращает текущую позицию
<code>final Buffer position(int n)</code>	Задает текущую позицию буфера равной <i>n</i> . Возвращает ссылку на буфер
<code>int remaining()</code>	Возвращает количество элементов, доступных до того, как будет достигнут предел. Иными словами, возвращается предел минус текущая позиция
<code>final Buffer reset()</code>	Устанавливает текущую позицию в вызывающем буфере на установленной ранее метке. Возвращает ссылку на буфер
<code>final Buffer rewind()</code>	Устанавливает текущую позицию в вызывающем буфере в нуль. Возвращает ссылку на буфер

От класса `Buffer` происходят приведенные ниже классы конкретных буферов, где тип хранимых данных можно определить по их именам. Класс `MappedByteBuffer` является производным от класса `ByteBuffer` и используется для сопоставления файла с буфером.

<code>ByteBuffer</code>	<code>CharBuffer</code>	<code>DoubleBuffer</code>	<code>FloatBuffer</code>
<code>IntBuffer</code>	<code>LongBuffer</code>	<code>MappedByteBuffer</code>	<code>ShortBuffer</code>

Все упомянутые выше буфера предоставляют различные методы `get()` и `put()`, которые позволяют получать данные из буфера или вносить их в него. (Разумеется, метод `put()` недоступен, если буфер предназначен только для чтения.) В табл. 21.3 перечислены методы `get()` и `put()`, определенные в классе `ByteBuffer`. Другие классы буферов имеют похожие методы. Во всех классах буферов поддерживают-

ся также методы, выполняющие различные операции с буфером. Например, с помощью метода `allocate()` можно вручную выделить оперативную память под буфер, с помощью метода `wrap()` — организовать массив в пределах буфера, а с помощью метода `slice()` — создать подпоследовательность в буфере.

**Таблица 21.3. Методы `get()` и `put()` из класса `ByteBuffer`**

Метод	Описание
<code>abstract byte get()</code>	Возвращает байт на текущей позиции
<code>ByteBuffer get(byte значения[])</code>	Копирует вызывающий буфер в заданный массив <i>значения</i> . Возвращает ссылку на буфер. Если же в буфере не осталось больше элементов, количество которых равно <i>значения.length</i> , то генерируется исключение типа <code>BufferUnderflowException</code>
<code>ByteBuffer get(byte значения[], int начало, int количество)</code>	Копирует заданное количество элементов из вызывающего буфера в указанный массив <i>значения</i> , начиная с позиции по индексу <i>начало</i> . Возвращает ссылку на буфер. Если в буфере больше не осталось заданное количество элементов, то генерируется исключение типа <code>BufferUnderflowException</code>
<code>abstract byte get(int индекс)</code>	Возвращает из вызывающего буфера байт по указанному <i>индексу</i>
<code>abstract ByteBufferput(byte b)</code>	Копирует заданный байт <i>b</i> на текущую позицию в вызывающем буфере. Возвращает ссылку на буфер. Если буфер заполнен, то генерируется исключение типа <code>BufferOverflowException</code>
<code>final ByteBufferput(byte значения[])</code>	Копирует все элементы из указанного массива <i>значения</i> в вызывающий буфер, начиная с текущей позиции. Возвращает ссылку на буфер. Если буфер не может вместить все элементы, то генерируется исключение типа <code>BufferOverflowException</code>
<code>ByteBuffer put(byte значения[], int начало, int количество)</code>	Копирует в вызывающий буфер заданное количество элементов из указанного массива <i>значения</i> , начиная с указанной позиции <i>начало</i> . Возвращает ссылку на буфер. Если буфер не может хранить все элементы, то генерируется исключение типа <code>BufferOverflowException</code>
<code>ByteBufferput(ByteBuffer bb)</code>	Копирует элементы из заданного буфера <i>bb</i> в вызывающий буфер, начиная с текущей позиции. Если буфер не может хранить все элементы, то генерируется исключение типа <code>BufferOverflowException</code> . Возвращает ссылку на буфер
<code>abstract ByteBufferput(int индекс, byte b)</code>	Копирует байт <i>b</i> на позицию по указанному <i>индексу</i> в вызывающем буфере. Возвращает ссылку на буфер

## Каналы

Каналы определены в пакете `java.nio.channels`. *Канал* представляет открытое соединение с источником или адресатом ввода-вывода. Классы каналов реализуют ин-

терфейс `Channel`, расширяющий интерфейс `Closeable`, а начиная с JDK 7 – интерфейс `AutoCloseable`. При реализации интерфейса `AutoCloseable` каналами можно управлять в блоке оператора `try` с ресурсами, где канал закрывается автоматически, когда он больше не нужен. (Подробнее об операторе `try` с ресурсами см. в главе 13.)

Один из способов получения канала подразумевает вызов метода `getChannel()` для объекта, поддерживающего каналы. Например, метод `getChannel()` поддерживается в следующих классах ввода-вывода:

<code>DatagramSocket</code>	<code>FileInputStream</code>	<code>FileOutputStream</code>
<code>RandomAccessFile</code>	<code>ServerSocket</code>	<code>Socket</code>

Конкретный тип возвращаемого канала зависит от типа объекта, для которого вызывается метод `getChannel()`. Например, когда метод `getChannel()` вызывается для объекта типа `FileInputStream`, `FileOutputStream` или `RandomAccessFile`, он возвращает канал типа `FileChannel`. А если этот метод вызывается для объекта типа `Socket`, то он возвращает канал типа `SocketChannel`.

Еще один способ получения канала подразумевает использование одного из статических методов, определенных в классе `Files`, который был введен в версии JDK 7. Например, используя класс `Files`, можно получить байтовый канал при вызове метода `newByteChannel()`. Он возвращает канал типа `SeekableByteChannel`, т.е. интерфейса, реализуемого классом `FileChannel`. (Более подробно класс `Files` рассматривается далее в этой главе.)

В каналах типа `FileChannel` и `SocketChannel` поддерживаются различные методы `read()` и `write()`, которые позволяют выполнять операции ввода-вывода через канал. Например, в табл. 21.4 перечислен ряд методов `read()` и `write()`, определенных в классе `FileChannel`.

**Таблица 21.4. Методы `read()` и `write()` из класса `FileChannel`**

Метод	Описание
<code>abstract int read(ByteBuffer bb) throws IOException</code>	Считывает байты из вызывающего канала в указанный буфер <i>bb</i> до тех пор, пока буфер не будет заполнен или же не исчерпятся вводимые данные. Возвращает количество прочитанных байтов
<code>abstract int read(ByteBuffer bb, long начало) throws IOException</code>	Считывает байты из вызывающего канала в указанный буфер <i>bb</i> , начиная с позиции <i>начало</i> и до тех пор, пока буфер не будет заполнен или же не исчерпятся вводимые данные. Текущая позиция не изменяется. Возвращает количество прочитанных байтов или значение <code>-1</code> , если позиция <i>начало</i> окажется за пределами файла
<code>abstract int write(ByteBuffer bb) throws IOException</code>	Записывает содержимое байтового буфера в вызывающий канал, начиная с текущей позиции. Возвращает количество записанных байтов
<code>abstract int write(ByteBuffer bb, long начало) throws IOException</code>	Записывает содержимое байтового буфера в вызывающий канал, начиная с позиции <i>начало</i> в файле. Текущая позиция не изменяется. Возвращает количество записанных байтов

Все каналы поддерживают дополнительные методы, предоставляющие доступ к каналу и позволяющие управлять им. Например, канал типа `FileChannel` поддерживает среди прочего методы для получения и установки текущей позиции, передачи данных между файловыми каналами, получения текущего размера канала и его блокировки. В классе `FileChannel` предоставляется статический метод `open()`, который открывает файл и возвращает для него канал. Такой результат достигается другим способом получения канала. В классе `FileChannel` предоставляется также метод `map()`, с помощью которого можно сопоставить файл с буфером.

## Наборы символов и селекторы

В системе ввода-вывода NIO применяются наборы символов и селекторы. *Набор символов* определяет способ сопоставления байтов с символами. С помощью *кодера* можно закодировать последовательность символов в виде байтов. Процесс декодирования производится с помощью *декодера*. Наборы символов, кодеры и декодеры поддерживаются в классах, определяемых в пакете `java.nio.charset`. Кодеры и декодеры предоставляются по умолчанию, и поэтому обращаться непосредственно к наборам символов приходится крайне редко.

*Селектор* обеспечивает возможность многоканального ввода-вывода по ключам, не прибегая к блокировке. Иными словами, с помощью селекторов можно выполнять операции ввода-вывода через несколько каналов. Селекторы поддерживаются классами, определяемыми в пакете `java.nio.channels`. Они чаще всего применяются в каналах, опирающихся на сокет. В примерах, представленных в этой главе, наборы символов и селекторы не применяются, тем не менее они могут оказаться полезными в ряде приложений.

## Усовершенствования в системе NIO, начиная с версии JDK 7

В версии JDK 7 система ввода-вывода NIO была значительно расширена и усовершенствована. Помимо поддержки оператора `try` с ресурсами, который обеспечивает автоматическое управление ресурсами, усовершенствования включают три новых пакета (`java.nio.file`, `java.nio.file.attribute` и `java.nio.file.spi`), несколько новых классов, интерфейсов и методов, а также прямую поддержку потокового ввода-вывода. Эти усовершенствования существенно расширили возможности для применения системы ввода-вывода NIO, особенно в файлы. В последующих разделах описывается ряд ключевых дополнений данной системы.

## Интерфейс `Path`

Возможно, одним из наиболее важных дополнений системы ввода-вывода NIO является интерфейс `Path`, поскольку он инкапсулирует путь к файлу. Как будет показано далее, интерфейс `Path` служит связующим звеном для большинства новых файловых средств в системе ввода-вывода NIO. Он описывает расположе-

ние файла в структуре каталогов. Интерфейс `Path` находится в пакете `java.nio.file` и наследует интерфейсы `Watchable`, `Iterable<Path>` и `Comparable<Path>`. Интерфейс `Watchable` описывает объект, который можно наблюдать и изменять. Интерфейсы `Iterable` и `Comparable` были представлены ранее в данной книге.

В интерфейсе `Path` объявляется немало методов для манипулирования путями к файлам. Некоторые из них приведены в табл. 21.5. Обратите особое внимание на метод `getName()`. Он служит для получения элемента пути. С этой целью в данном методе применяется индекс. Нулевому значению индекса соответствует ближайшая к корневому каталогу часть пути, являющаяся его крайним слева элементом. Последующие индексы определяют элементы вправо от корневого каталога. Количество элементов в пути может быть получено в результате вызова метода `getNameCount()`. Если же требуется получить строковое представление всего пути, достаточно вызвать метод `toString()`. Следует также заметить, что для распознавания относительного и абсолютного пути достаточно вызвать метод `resolve()`.

**Таблица 21.5. Избранные методы из интерфейса `Path`**

Метод	Описание
<code>boolean endsWith(String путь)</code>	Возвращает логическое значение <b>true</b> , если вызывающий объект типа <b>Path</b> оканчивается путем, определяемым параметром <i>путь</i> , а иначе — логическое значение <b>false</b>
<code>boolean endsWith(Path путь)</code>	Возвращает логическое значение <b>true</b> , если вызывающий объект типа <b>Path</b> оканчивается путем, определяемым параметром <i>путь</i> , а иначе — логическое значение <b>false</b>
<code>Path getFileName()</code>	Возвращает имя файла, связанное с вызывающим объектом типа <b>Path</b>
<code>Path getName(int индекс)</code>	Возвращает объект типа <b>Path</b> , содержащий имя элемента пути по указанному <i>индексу</i> в вызывающем объекте. Крайний слева элемент имеет нулевой индекс и находится ближе всего к корневому каталогу. А крайний справа элемент имеет индекс <code>getNameCount() - 1</code>
<code>int getNameCount()</code>	Возвращает количество элементов (кроме корневого) в вызывающем объекте типа <b>Path</b>
<code>Path getParent()</code>	Возвращает объект типа <b>Path</b> , который содержит весь путь, кроме имени файла, определяемого вызывающим объектом типа <b>Path</b>
<code>Path getRoot()</code>	Возвращает корневой каталог из вызывающего объекта типа <b>Path</b>
<code>boolean isAbsolute()</code>	Возвращает логическое значение <b>true</b> , если вызывающий объект типа <b>Path</b> обозначает абсолютный путь, а иначе — логическое значение <b>false</b>

Метод	Описание
<code>Path resolve(Path путь)</code>	Если указанный <i>путь</i> является абсолютным, то возвращается именно он. А если указанный <i>путь</i> не содержит корневой каталог, то этот <i>путь</i> предваряется корневым каталогом из вызывающего объекта типа <b>Path</b> , а затем возвращается полученный результат. Если же указанный <i>путь</i> пуст, то возвращается вызывающий объект типа <b>Path</b> . В противном случае поведение данного метода не определено
<code>Path resolve(String путь)</code>	Если указанный <i>путь</i> является абсолютным, возвращается именно этот <i>путь</i> . А если указанный <i>путь</i> не содержит корневой каталог, то этот <i>путь</i> предваряется корневым каталогом из вызывающего объекта типа <b>Path</b> , а затем возвращается полученный результат. Если же указанный <i>путь</i> пуст, то возвращается вызывающий объект типа <b>Path</b> . В противном случае поведение данного метода не определено
<code>boolean startsWith(String путь)</code>	Возвращает логическое значение <b>true</b> , если вызывающий объект типа <b>Path</b> начинается с указанного <i>пути</i> , а иначе — логическое значение <b>false</b>
<code>boolean startsWith(Path путь)</code>	Возвращает логическое значение <b>true</b> , если вызывающий объект типа <b>Path</b> начинается с указанного <i>пути</i> , а иначе — логическое значение <b>false</b>
<code>Path toAbsolutePath()</code>	Возвращает вызывающий объект типа <b>Path</b> в виде абсолютного пути
<code>String toString()</code>	Возвращает строковое представление вызывающего объекта типа <b>Path</b>

Следует также иметь в виду, что при обновлении унаследованного кода, в котором используется класс `File`, определенный в пакете `java.io`, экземпляр класса `File` можно преобразовать в экземпляр интерфейса `Path`, вызвав метод `toPath()` для объекта типа `File`. Этот метод был введен в класс `File` в версии JDK 7. Кроме того, экземпляр класса `File` можно получить, вызвав метод `toFile()`, определяемый в интерфейсе `Path`.

## Класс `Files`

Большинство действий, которые выполняются над файлами, предоставляются статическими методами из класса `Files`. Путь к файлу, над которым выполняются определенные действия, задает объект типа `Path`. Таким образом, методы из класса `Files` используют объект типа `Path`, чтобы указать используемый файл. Класс `Files` обладает обширным рядом функциональных возможностей. Так, в нем имеются методы, позволяющие открывать или создавать файл по указанному пути. Кроме того, из объекта типа `Path` можно получить следующие сведения о файле: является ли он исполняемым, скрытым или доступным только для чтения. В классе

Files предоставляются также методы, позволяющие копировать или перемещать файлы. Некоторые методы, определенные в этом классе, перечислены в табл. 21.6. Помимо исключения типа IOException, возможны и другие исключения. В версии JDK 8 класс Files дополнен следующими четырьмя методами: list(), walk(), lines() и find(). Все эти методы возвращают объект типа Stream. Они способствуют интеграции системы ввода-вывода NIO с новым прикладным программным интерфейсом API потоков ввода-вывода, определенным в версии JDK 8 и описываемым в главе 29.

Таблица 21.6. Избранные методы из класса Files

Метод	Описание
<code>static Path copy(Path источник, Path адресат CopyOption ... способ) throws IOException</code>	Копирует файл из <i>источника</i> по указанному <i>адресату</i> заданным <i>способом</i>
<code>static Path createDirectory(Path путь, FileAttribute&lt;?&gt; ... атрибуты) throws IOException</code>	Создает каталог по указанному <i>пути</i> . Атрибуты каталога определяются параметром <i>атрибуты</i>
<code>static Path createFile(Path путь, FileAttribute&lt;?&gt; ... атрибуты) throws IOException</code>	Создает файл по указанному <i>пути</i> . Атрибуты файла определяются параметром <i>атрибуты</i>
<code>static void delete(Path путь) throws IOException</code>	Удаляет файл по указанному <i>пути</i>
<code>static boolean exists(Path путь, LinkOptions ... параметры)</code>	Возвращает логическое значение <b>true</b> , если файл существует по указанному <i>пути</i> , а иначе — логическое значение <b>false</b> . Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент <i>параметры</i> должен принимать значение <b>NOFOLLOW_LINKS</b>
<code>static boolean isDirectory(Path путь, LinkOptions ... параметры)</code>	Возвращает логическое значение <b>true</b> , если параметр <i>путь</i> определяет каталог, а иначе — логическое значение <b>false</b> . Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам заданный аргумент <i>параметры</i> должен принимать значение <b>NOFOLLOW_LINKS</b>
<code>static boolean isExecutable(Path путь)</code>	Возвращает логическое значение <b>true</b> , если файл по указанному <i>пути</i> является исполняемым, а иначе — логическое значение <b>false</b>
<code>static boolean isHidden(Path путь) throws IOException</code>	Возвращает логическое значение <b>true</b> , если файл по указанному <i>пути</i> является скрытым, а иначе — логическое значение <b>false</b>
<code>static boolean isReadable(Path путь)</code>	Возвращает логическое значение <b>true</b> , если файл по указанному <i>пути</i> доступен для чтения, а иначе — логическое значение <b>false</b>

Метод	Описание
<pre>static boolean isRegularFile(Path <i>путь</i>, LinkOptions ... <i>параметры</i>)</pre>	<p>Возвращает логическое значение <b>true</b>, если параметр <i>путь</i> определяет файл, а иначе — логическое значение <b>false</b>. Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент <i>параметры</i> должен принимать значение <b>NOFOLLOW_LINKS</b></p>
<pre>static boolean isWritable(Path <i>путь</i>)</pre>	<p>Возвращает логическое значение <b>true</b>, если файл по указанному <i>пути</i> доступен для записи, а иначе — логическое значение <b>false</b></p>
<pre>static Path move(Path <i>источник</i>, Path <i>адресат</i>, CopyOption ... <i>способ</i>) throws IOException</pre>	<p>Копирует файл из <i>источника</i> по указанному <i>адресату</i> заданным <i>способом</i></p>
<pre>static SeekableByteChannel newByteChannel(Path <i>путь</i>, OpenOption ... <i>способ</i>) throws IOException</pre>	<p>Открывает файл по указанному <i>пути</i> заданным <i>способом</i>. Возвращает для файла байтовый канал типа <b>SeekableByteChannel</b>. Текущая позиция в этом канале может быть изменена. Интерфейс <b>SeekableByteChannel</b> реализуется классом <b>FileChannel</b></p>
<pre>static DirectoryStream&lt;Path&gt; newDirectoryStream(Path <i>путь</i>) throws IOException</pre>	<p>Открывает каталог по указанному <i>пути</i>. Возвращает поток ввода каталога типа <b>DirectoryStream</b>, связанный с каталогом</p>
<pre>static InputStream newInputStream(Path <i>путь</i>, OpenOption ... <i>способ</i>) throws IOException</pre>	<p>Открывает файл по указанному <i>пути</i> заданным <i>способом</i>. Возвращает поток ввода типа <b>InputStream</b>, связанный с файлом</p>
<pre>static OutputStream newOutputStream(Path <i>путь</i>, OpenOption ... <i>способ</i>) throws IOException</pre>	<p>Открывает файл по указанному <i>пути</i> заданным <i>способом</i>. Возвращает поток вывода типа <b>OutputStream</b>, связанный с файлом</p>
<pre>static boolean notExists(Path <i>путь</i>, LinkOption ... <i>параметры</i>)</pre>	<p>Возвращает логическое значение <b>true</b>, если файл по указанному <i>пути</i> не существует, а иначе — логическое значение <b>false</b>. Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент <i>параметры</i> должен принимать значение <b>NOFOLLOW_LINKS</b></p>
<pre>static &lt;A extends BasicFileAttributes&gt; A readAttributes(Path <i>путь</i>, Class&lt;A&gt; <i>тип_атрибута</i>, LinkOption ... <i>параметры</i>) throws IOException</pre>	<p>Получает атрибуты, связанные с файлом. Тип передаваемых атрибутов определяется параметром <i>тип_атрибута</i>. Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент <i>параметры</i> должен принимать значение <b>NOFOLLOW_LINKS</b></p>
<pre>static long size(Path <i>путь</i>) throws IOException</pre>	<p>Возвращает размер файла по указанному <i>пути</i></p>

Обратите внимание на то, что некоторые методы, перечисленные в табл. 21.6, получают аргумент типа `OpenOption`. Это интерфейс, описывающий способ открытия файла. Он реализуется классом `StandardOpenOption`, где определяется перечисление, значения которого представлены в табл. 21.7.

**Таблица 21.7. Стандартные значения параметров открытия файлов**

Значение	Назначение
<code>APPEND</code>	Присоединить выводимые данные в конце файла
<code>CREATE</code>	Создать файл, если он еще не существует
<code>CREATE_NEW</code>	Создать файл только в том случае, если он еще не существует
<code>DELETE_ON_CLOSE</code>	Удалить файл, когда он закрывается
<code>DSYNC</code>	Немедленно записать вносимые изменения в физический файл. Как правило, для повышения производительности изменения в файле буферизируются файловой системой и записываются только по мере надобности
<code>READ</code>	Открыть файл для операций ввода
<code>SPARSE</code>	Указать файловой системе, что файл разрежен, а следовательно, не может быть полностью заполнен данными. Если файловая система не поддерживает разреженные файлы, это значение параметра игнорируется
<code>SYNC</code>	Немедленно записать вносимые изменения в файл или его метаданные в физический файл. Как правило, для повышения производительности изменения в файле буферизируются файловой системой и записываются только по мере надобности
<code>TRUNCATE_EXISTING</code>	Укоротить до нуля длину уже существующего файла, открываемого для вывода
<code>WRITE</code>	Открыть файл для операций вывода

## Класс `Paths`

Экземпляр типа `Path` нельзя создать непосредственно с помощью конструктора, поскольку это интерфейс, а не класс. Вместо этого можно получить объект типа `Path`, вызвав метод, который возвращает этот объект. Как правило, для этой цели служит метод `get()`, определяемый в классе `Paths`. Существуют две формы метода `get()`. Ниже приведена та его форма, которая употребляется в примерах этой главы.

```
static Path get(String имя_пути, String ... части)
```

Этот метод возвращает объект, инкапсулирующий определенный путь. Путь может быть задан двумя способами. Если параметр `части` не указан, то путь должен полностью определяться параметром `имя_пути`. В качестве альтернативы путь можно передать по частям, причем первую часть в качестве параметра `имя_пути`, а остальные части — в качестве параметра `части` переменной длины. Но в любом случае метод `get()` сгенерирует исключение типа `InvalidPathException`, если указанный путь синтаксически недостоверен.

Во второй форме метода `get()` объект типа `Path` создается из URI. Эта форма выглядит следующим образом:

```
static Path get(URI uri)
```

В итоге возвращается объект типа `Path`, соответствующий заданному параметру `uri`. Следует, однако, иметь в виду, что создание объекта типа `Path` не приводит к открытию или созданию файла. Вместо этого лишь создается объект, инкапсулирующий путь к каталогу, в котором находится файл.

## Интерфейсы атрибутов файлов

С файлами связан ряд атрибутов, обозначающих время создания файла, время его последней модификации, размер файла или каталог. Система ввода-вывода NIO организует атрибуты файлов в виде иерархии различных интерфейсов, определенных в пакете `java.nio.file.attribute`. На вершине этой иерархии находится интерфейс `BasicFileAttributes`, инкапсулирующий ряд атрибутов, которые обычно применяются в большинстве файловых систем. Методы, определенные в интерфейсе `BasicFileAttributes`, перечислены в табл. 21.8.

**Таблица 21.8. Методы из интерфейса `BasicFileAttributes`**

Метод	Описание
<code>FileTime creationTime()</code>	Возвращает время создания файла. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
<code>Object fileKey()</code>	Возвращает файловый ключ. Если этот атрибут не поддерживается файловой системой, возвращается пустое значение <code>null</code>
<code>boolean isDirectory()</code>	Возвращает логическое значение <code>true</code> , если файл является каталогом
<code>boolean isOther()</code>	Возвращает логическое значение <code>true</code> , если файл является символической ссылкой или каталогом, а не файлом
<code>boolean isRegularFile()</code>	Возвращает логическое значение <code>true</code> , если файл является обычным файлом, а не каталогом или символической ссылкой
<code>boolean isSymbolicLink()</code>	Возвращает логическое значение <code>true</code> , если файл является символической ссылкой
<code>FileTime lastAccessTime()</code>	Возвращает время последнего обращения к файлу. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
<code>FileTime lastModifiedTime()</code>	Возвращает время последней модификации файла. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
<code>long size()</code>	Возвращает размер файла

Производными от интерфейса `BasicFileAttributes` являются следующие два интерфейса: `DosFileAttributes` и `PosixFileAttributes`. В частности, интерфейс `DosFileAttributes` описывает атрибуты, связанные с файловой системой FAT, которые были первоначально определены в файловой системе DOS. В этом интерфейсе определяются методы, перечисленные в табл. 21.9.

**Таблица 21.9. Методы из интерфейса `DosFileAttributes`**

Метод	Описание
<code>boolean isArchive()</code>	Возвращает логическое значение <b>true</b> , если файл помечен как архивный, а иначе — логическое значение <b>false</b>
<code>boolean isHidden()</code>	Возвращает логическое значение <b>true</b> , если файл помечен как скрытый, а иначе — логическое значение <b>false</b>
<code>boolean isReadOnly()</code>	Возвращает логическое значение <b>true</b> , если файл помечен как доступный только для чтения, а иначе — логическое значение <b>false</b>
<code>boolean isSystem()</code>	Возвращает логическое значение <b>true</b> , если файл помечается как системный, а иначе — логическое значение <b>false</b>

Интерфейс `PosixFileAttributes` инкапсулирует атрибуты, определенные по стандартам POSIX (Portable Operating System Interface — переносимый интерфейс операционных систем). В этом интерфейсе определяются методы, перечисленные в табл. 21.10.

**Таблица 21.10. Методы из интерфейса `PosixFileAttributes`**

Метод	Описание
<code>GroupPrincipal group()</code>	Возвращает группового владельца файла
<code>UserPrincipal owner()</code>	Возвращает отдельного владельца файла
<code>Set&lt;PosixFilePermission&gt; permissions()</code>	Возвращает полномочия доступа к файлу

Имеются разные способы доступа к атрибутам файлов. В частности, вызвав статический метод `readAttributes()`, определенный в классе `Files`, можно получить объект, инкапсулирующий атрибуты файла. Ниже приведена одна из общих форм объявления этого метода.

```
static <A extends BasicFileAttributes>
    A readAttributes(Path путь, Class<A> тип_атрибута,
        LinkOption... параметры) throws IOException
```

Этот метод возвращает ссылку на объект, обозначающий атрибуты файла по указанному пути. Конкретный тип атрибутов указывается в виде объекта типа `Class` с помощью параметра `тип_атрибута`. Например, для получения основных атрибутов файла следует передать в качестве параметра `тип_атрибута` объект типа `BasicFileAttributes.class`, для получения атрибутов DOS — объект типа `DosFileAttributes.class`, а для получения атрибутов POSIX — объект типа `PosixFileAttributes.class`. Дополнительные, но необязательные параметры ссылок передаются как аргумент `параметры`. Если же аргумент `параметры` не опре-

делен, то следуют символические ссылки. Метод `readAttributes()` возвращает ссылку на требуемый атрибут. Если же тип требуемого атрибута недоступен, то генерируется исключение типа `UnsupportedOperationException`. Используя объект, возвращаемый этим методом, можно обратиться к атрибутам файла.

Еще один способ доступа к атрибутам файла состоит в том, чтобы вызвать метод `getFileAttributeView()`, определенный в классе `Files`. В системе ввода-вывода NIO определяется несколько интерфейсов для представлений атрибутов, в том числе `AttributeView`, `BasicFileAttributeView`, `DosFileAttributeView` и `PosixFileAttributeView`. В примерах из этой главы представления атрибутов не употребляются, но в некоторых случаях это средство может оказаться полезным.

Иногда можно и не прибегать непосредственно к интерфейсам атрибутов файлов, поскольку в классе `Files` предоставляются служебные статические методы, позволяющие обращаться к некоторым атрибутам. Для этой цели в классе `Files` имеются такие методы, как `isHidden()` и `isWritable()`.

Следует, однако, иметь в виду, что все допустимые атрибуты файлов поддерживаются не во всех файловых системах. Например, атрибуты файлов DOS относятся к файловой системе FAT, хотя первоначально они были определены в файловой системе DOS. Те атрибуты, которые применяются в обширном ряде файловых систем, описаны в интерфейсе `BasicFileAttributes`. Поэтому именно они и употребляются в примерах из этой главы.

## Классы `FileSystem`, `FileSystems` и `FileStore`

Для упрощения доступа к файловой системе в пакете `java.nio.file` предоставляются классы `FileSystem` и `FileSystems`. В действительности, используя метод `newFileSystem()`, определенный в классе `FileSystems`, можно даже получить новую файловую систему. А класс `FileStore` инкапсулирует систему хранения файлов. И хотя упоминаемые здесь классы не употребляются в примерах из этой главы, им можно найти применение в своих прикладных программах.

## Применение системы ввода-вывода NIO

В этом разделе демонстрируется применение системы ввода-вывода NIO для решения самых разных задач. Но прежде следует подчеркнуть, что в версии JDK 7 была значительно расширена как сама система ввода-вывода NIO, так и область ее применения. Как упоминалось ранее, усовершенствованную версию этой системы иногда называют NIO.2. Вследствие столь значительных усовершенствований в системе ввода-вывода NIO.2 изменился и способ написания кода на ее основе, а также расширился круг задач, для решения которых можно ее применять. В силу этого обстоятельства в большей части примеров из этой главы применяются средства из системы ввода-вывода NIO.2, и поэтому для проработки этих примеров потребуется комплект JDK 7, JDK 8 или более поздняя его версия. Тем не менее в конце этой главы дается краткое описание кода, написанного до версии JDK 7, в помощь тем программистам, которые пользуются системой ввода-вывода до версии JDK 7 или сопровождают унаследованный код.

**Помните!** Для компиляции большинства примеров из этой главы требуется комплект JDK 7, JDK 8 или более поздняя его версия.

Раньше система NIO предназначалась в основном для канального ввода-вывода, и эта разновидность ввода-вывода по-прежнему остается важнейшей областью ее применения. Но теперь систему ввода-вывода NIO можно также использовать для потокового ввода-вывода и выполнения операций в файловой системе. Таким образом, обсуждение областей применения системы ввода-вывода NIO можно разделить на три части:

- канальный ввод-вывод;
- потоковый ввод-вывод;
- операции в файловой системе.

Самым распространенным средством ввода-вывода является файл на диске, поэтому в примерах, представленных далее в этой главе, употребляются файлы на диске. А поскольку все канальные операции ввода-вывода в файлы основываются на передаче байтов, то для их выполнения будут использоваться буфера типа `ByteBuffer`.

Прежде чем открыть файл для доступа к нему средствами системы ввода-вывода NIO, следует получить объект типа `Path`, описывающий этот файл. Это можно, в частности, сделать, вызвав упоминавшийся ранее фабричный метод `Paths.get()`. В приведенных далее примерах употребляется следующая общая форма объявления метода `get()`:

```
static Path get(String имя_пути, String ... части)
```

Напомним, что путь к файлу можно указать двумя способами. Во-первых, передать первую его часть в качестве параметра `имя_пути`, а остальные части — в качестве параметра `части` переменной длины. И во-вторых, указать весь путь в качестве параметра `имя_пути`, а параметр `части` опустить. Именно такой способ и применяется в рассматриваемых далее примерах.

## Применение системы NIO для канального ввода-вывода

Важнейшей областью применения системы ввода-вывода NIO является получение доступа к файлу через каналы и буфера. В последующих разделах демонстрируются некоторые способы применения канала для чтения и записи данных в файл.

### Чтение файла через канал

Имеется несколько способов чтения данных из файла через канал. Наиболее распространенный из них, вероятно, состоит в том, чтобы сначала выделить оперативную память под буфер вручную, а затем выполнить явным образом операцию чтения для загрузки этого буфера данными из файла. Поэтому рассмотрим этот способ в первую очередь.

Прежде чем прочитать данные из файла, его нужно открыть. Для этого сначала создается объект типа `Path`, описывающий файл, а затем он используется для открытия файла. Имеются разные способы открыть файл в зависимости от того, как он будет использоваться. В рассматриваемом здесь примере файл будет открыт для выполнения явных операций байтового ввода. Поэтому в данном примере для открытия файла и установления канала доступа к нему вызывается метод `Files.newByteChannel()`. Метод `newByteChannel()` имеет следующую общую форму:

```
static SeekableByteChannel newByteChannel(
    Path путь, OpenOption ... способ)
    throws IOException
```

Этот метод возвращает объект типа `SeekableByteChannel`, инкапсулирующий канал для файловых операций. Объект типа `Path`, описывающий файл, передается в качестве параметра *путь*. А параметр *способ* определяет порядок открытия файла. Это параметр переменной длины, и поэтому в качестве его можно указать любое количество аргументов через запятую. (Допустимые значения параметров данного метода обсуждались ранее и приведены в табл. 21.7.) Если же никаких аргументов не указано, то файл открывается для операций ввода. Интерфейс `SeekableByteChannel` описывает канал, применяемый для файловых операций. Он реализуется классом `FileChannel`. Когда используется выбираемая по умолчанию файловая система, возвращаемый объект может быть приведен к типу `FileChannel`. Завершив работу с каналом, следует закрыть его. Классы всех каналов, включая и класс `FileChannel`, реализуют интерфейс `AutoCloseable`, поэтому для автоматического закрытия файла вместо явного вызова метода `close()` можно воспользоваться оператором `try` с ресурсами. Именно такой подход и применяется в примере из этой главы.

Затем следует получить буфер, который будет использоваться каналом, заключив буфер в оболочку существующего массива или динамически выделив оперативную память под буфер. В примерах из этой главы применяется выделение оперативной памяти под буфер, но вы вольны выбрать любой из этих двух способов. Файловые каналы оперируют буферами байтов, и поэтому для их получения в примерах из этой главы вызывается метод `allocate()`, определенный в классе `ByteBuffer`. Ниже приведена его общая форма, где *емкость* обозначает конкретную емкость буфера, а в итоге возвращается ссылка на буфер.

```
static ByteBuffer allocate(int емкость)
```

После создания буфера для канала вызывается метод `read()`, которому передается ссылка на буфер. Ниже приведена общая форма метода `read()`, которая употребляется в примерах, представленных далее в главе.

```
int read(ByteBuffer буфер) throws IOException
```

При каждом вызове метода `read()` указанный *буфер* заполняется данными из файла. Чтение осуществляется последовательно, а следовательно, при каждом вызове метода `read()` из файла в буфер читается следующая порция байтов. Метод `read()` возвращает количество фактически прочитанных байтов. При попытке прочитать данные по достижении конца файла возвращается значение `-1`.

В приведенном ниже примере программы все изложенное выше демонстрируется на практике. В этой программе данные из файла `test.txt` читаются через канал посредством явных операций ввода.

```
// Использовать канал ввода-вывода для чтения файла.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;
        Path filepath = null;

        // сначала получить путь к файлу
        try {
            filepath = Paths.get("test.txt");
        } catch (InvalidPathException e) {
            System.out.println("Path Error " + e);
            return;
        }

        // затем получить канал к этому файлу в
        // блоке оператора try с ресурсами
        try (SeekableByteChannel fChan = Files.newByteChannel(filepath))
        {

            // выделить память под буфер
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
                // читать данные из файла в буфер
                count = fChan.read(mBuf);

                // прекратить чтение по достижении конца файла
                if(count != -1) {

                    // подготовить буфер к чтению из него данных
                    mBuf.rewind();

                    // читать байты данных из буфера и
                    // выводить их на экран как символы
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);

            System.out.println();
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

Эта программа действует следующим образом. Сначала создается объект типа `Path`, содержащий относительный путь к файлу `test.txt`. Ссылка на этот объект присваивается переменной `filepath`. Затем для создания канала, связанного с фай-

лом, вызывается метод `newByteChannel()`, которому передается ссылка на файл в переменной `filepath`. А поскольку никаких параметров открытия файла не указано, то файл по умолчанию открывается для чтения. Обратите внимание на то, что созданный в итоге канал является объектом, управляемым оператором `try` с ресурсами. Таким образом, канал автоматически закрывается в конце блока этого оператора. Далее в данной программе вызывается метод `allocate()` из класса `ByteBuffer`, чтобы выделить оперативную память под буфер для хранения содержимого файла во время чтения. Ссылка на этот буфер хранится в переменной экземпляра `mBuf`. После этого вызывается метод `read()`, и содержимое файла читается по очереди в буфер `mBuf`. Количество прочитанных байтов сохраняется в переменной `count`. Затем вызывается метод `rewind()`, чтобы подготовить буфер к чтению из него данных. Этот метод нужно вызвать потому, что после вызова метода `read()` текущая позиция находится в конце буфера. Ее следует вернуть в начало буфера, чтобы при вызове метода `get()` можно было прочитать байты данных из буфера `mBuf`. (Напомним, что метод `get()` определяется в классе `ByteBuffer`.) Буфер `mBuf` может содержать только байты данных, поэтому из метода `get()` возвращаются байты. Они приводятся к типу `char`, чтобы выводить на экран содержимое файла в текстовом виде. (В качестве альтернативы можно создать буфер, автоматически преобразующий байты в символы, а затем прочитать их из этого буфера.) По достижении конца файла метод `read()` возвращает значение `-1`. В таком случае программа завершается и канал автоматически закрывается.

Обратите внимание на следующий интересный момент: программа получает объект типа `Path` в пределах одного блока оператора `try`, а затем использует его в другом блоке оператора `try` для получения и манипулирования каналом, связанным с файлом по пути, который задается объектом типа `Path`. И хотя в таком подходе нет ничего плохого, во многих случаях его можно упростить, чтобы организовать ввод-вывод данных только в одном блоке `try`. В этом случае вызовы методов `Paths.get()` и `newByteChannel()` следуют друг за другом. В качестве примера ниже приведена переделанная версия программы из предыдущего примера, где применяется данный подход.

```
// Более компактный способ открытия канала.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;

        // Здесь канал открывается по пути, возвращаемому
        // методом Paths.get() в виде объекта типа Path.
        // Переменная filepath больше не нужна
        try (SeekableByteChannel fChan =
            Files.newByteChannel(Paths.get("test.txt"))) {
            {
                // выделить память под буфер
                ByteBuffer mBuf = ByteBuffer.allocate(128);

                do {
```

```

// читать данные из файла в буфер
count = fChan.read(mBuf);

// прекратить чтение по достижении конца файла
if(count != -1) {

    // подготовить буфер к чтению из него данных
    mBuf.rewind();

    // читать байты данных из буфера и
    // выводить их на экран как символы
    for(int i=0; i < count; i++)
        System.out.print((char)mBuf.get());
    }
} while(count != -1);
System.out.println();
} catch(InvalidPathException e) {
    System.out.println("Ошибка указания пути " + e);
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода " + e);
}
}
}
}

```

В этой версии программы переменная `filepath` больше не требуется, и оба исключения обрабатываются в одном и том же блоке оператора `try`. Такой подход компактнее, поэтому именно он и применяется в остальных примерах из этой главы. Разумеется, при написании прикладного кода возможны случаи, когда создание объекта типа `Path` должно быть отделено от получения канала. В подобных случаях применяется предыдущий подход.

Другой способ чтения файла подразумевает его сопоставление с буфером. Преимущество такого способа состоит в том, что буфер автоматически получает содержимое файла. Никаких явных операций чтения не требуется. Сопоставление и чтение содержимого файла осуществляется в ходе следующей общей процедуры. Сначала получается объект типа `Path`, инкапсулирующий файл, как описано ранее. Затем получается канал к этому файлу. С этой целью вызывается метод `Files.newByteChannel()`, которому передается объект типа `Path`, а тип возвращаемого из него объекта приводится к типу `FileChannel`. Как упоминалось ранее, метод `newByteChannel()` возвращает объект типа `SeekableByteChannel`. Если используется выбираемая по умолчанию файловая система, этот объект может быть приведен к типу `FileChannel`. Затем для канала вызывается метод `map()`, чтобы сопоставить этот канал с буфером. Метод `map()` определяется в классе `FileChannel`, поэтому и требуется приведение к типу `FileChannel`. Ниже приведена общая форма объявления метода `map()`.

```

MappedByteBuffer map(
    FileChannel.MapMode способ, long позиция, long размер)
    throws IOException

```

В методе `map()` данные из файла сопоставляются с буфером в оперативной памяти. Значение параметра *способ* определяет вид разрешенной операции. Этот параметр может принимать одно из следующих допустимых значений:

<code>MapMode.READ_ONLY</code>	<code>MapMode.READ_WRITE</code>	<code>MapMode.PRIVATE</code>
--------------------------------	---------------------------------	------------------------------

Для чтения данных из файла следует указать значение `MapMode.READ_ONLY`, а для чтения и записи данных в файл — значение `MapMode.READ_WRITE`. Выбор значения `MapMode.PRIVATE` приводит к созданию закрытой копии файла, чтобы внесенные в буфере изменения не повлияли на основной файл. Место для начала сопоставления в пределах файла определяется параметром *позиция*, а количество сопоставляемых байтов — параметром *размер*. Ссылка на буфер возвращается в виде объекта класса `MappedByteBuffer`, производного от класса `ByteBuffer`. Как только файл будет сопоставлен с буфером, его содержимое можно прочитать в файл из буфера. Такой способ демонстрируется в следующем примере программы:

```
// Использовать сопоставление для чтения данных из файла.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelRead {
    public static void main(String args[]) {

        // получить канал к файлу в блоке оператора try с ресурсами
        try ( FileChannel fChan =
            (FileChannel) Files.newByteChannel(Paths.get("test.txt")) )
        {

            // получить размер файла
            long fSize = fChan.size();

            // а теперь сопоставить файл с буфером
            MappedByteBuffer mBuf =
                fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

            // читать байты из буфера и выводить их на экран
            for(int i=0; i < fSize; i++)
                System.out.print((char)mBuf.get());

            System.out.println();

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

В данной программе сначала создается путь к файлу, обозначаемый объектом типа `Path`, а затем открывается файл с помощью метода `newByteChannel()`. Получаемый в итоге канал приводится к типу `FileChannel` и сохраняется в переменной экземпляра `fChan`. Затем в результате вызова метода `size()` для канала получается размер файла. Далее вызывается метод `map()` по ссылке `fChan` на объект канала, чтобы сопоставить весь файл с областью в памяти, выделяемой под буфер, а ссылка на буфер сохраняется в переменной экземпляра `mBuf`. Обратите внимание на то, что переменная `mBuf` объявляется как ссылка на объект типа `MappedByteBuffer`. Байты из буфера в переменной `mBuf` читаются непосредственно методом `get()`.

## Запись данных в файл через канал

Как и при чтении данных из файла, для записи данных в файл через канал имеется несколько способов. Рассмотрим сначала один из наиболее распространенных способов. Он предполагает выделение оперативной памяти под буфер вручную, запись в него данных, а затем выполнение явной операции записи этих данных в файл.

Прежде чем записать данные в файл, его следует открыть. Для этого нужно получить сначала объект типа `Path`, обозначающий путь к файлу, а затем использовать этот путь, чтобы открыть файл. В рассматриваемом здесь примере файл будет открыт для выполнения явных операций байтового ввода. Поэтому в данном примере для открытия файла и установления канала доступа к нему вызывается метод `Files.newByteChannel()`. Как показано в предыдущем разделе, общая форма метода `newByteChannel()` такова:

```
static SeekableByteChannel newByteChannel(
    Path путь, OpenOption ... способ)
    throws IOException
```

Этот метод возвращает объект типа `SeekableByteChannel`, инкапсулирующий канал для файловых операций. Чтобы открыть файл для вывода, в качестве параметра `способ` следует передать значение `StandardOpenOption.WRITE`. Если файл еще не существует и его нужно создать, то следует указать также значение `StandardOpenOption.CREATE`. (Другие доступные значения стандартных параметров открытия файлов перечислены в табл. 21.7.) Как пояснялось в предыдущем разделе, интерфейс `SeekableByteChannel` описывает канал, применяемый для файловых операций. Его реализует класс `FileChannel`. Когда используется выбираемая по умолчанию файловая система, возвращаемый объект может быть приведен к типу `FileChannel`. Завершив работу с каналом, следует закрыть его.

Один из способов записи данных в файл через канал подразумевает явные вызовы метода `write()`. Сначала получается объект типа `Path`, обозначающий путь к файлу, а затем для открытия этого файла вызывается метод `newByteChannel()` и возвращаемый результат приводится к типу `FileChannel`. Далее выделяется оперативная память под буфер байтов, в который записываются выводимые в файл данные. Прежде чем данные будут записаны в файл, для буфера следует вызвать метод `rewind()`, чтобы обнулить его текущую позицию. (Каждая операция вывода в буфер увеличивает его текущую позицию. Поэтому перед записью в файл ее следует вернуть в исходное положение.) Далее для канала вызывается метод `write()`, которому передается буфер. Вся эта процедура демонстрируется в следующем примере программы, где весь английский алфавит записывается в файл `test.txt`:

```
// Записать данные в файл средствами системы ввода-вывода NIO.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelWrite {
```

```

public static void main(String args[]) {

    // получить канал к файлу в блоке оператора try с ресурсами
    try ( FileChannel fChan = (FileChannel)
        Files.newByteChannel(Paths.get("test.txt"),
            StandardOpenOption.WRITE,
            StandardOpenOption.CREATE) )
    {
        // создать буфер
        ByteBuffer mBuf = ByteBuffer.allocate(26);

        // записать некоторое количество байтов в буфер
        for(int i=0; i<26; i++)
            mBuf.put((byte)('A' + i));

        // подготовить буфер к записи данных
        mBuf.rewind();

        // записать данные из буфера в выходной файл
        fChan.write(mBuf);

    } catch(InvalidPathException e) {
        System.out.println("Ошибка указания пути " + e);
    } catch (IOException e) {
        System.out.println("Ошибка ввода-вывода: " + e);
        System.exit(1);
    }
}
}

```

Следует отметить одну важную особенность данной программы. Как упоминалось ранее, после записи данных в буфер байтов `mBuf`, но перед их записью в файл для буфера `mBuf` вызывается метод `rewind()`. Это требуется для обнуления текущей позиции после записи данных в буфер `mBuf`. Не следует забывать, что после каждого вызова метода `put()` для буфера `mBuf` текущая позиция смещается. Поэтому текущую позицию необходимо вернуть в начало буфера, прежде чем вызывать метод `write()`. Если не сделать этого, метод `write()` не сумеет обнаружить в буфере никаких данных, посчитав, что их там вообще нет.

Еще один способ обнуления буфера между операциями ввода и вывода подразумевает вызов метода `flip()` вместо метода `rewind()`. Метод `flip()` устанавливает для текущей позиции нулевое значение, а для предела — значение предыдущей текущей позиции. В приведенном выше примере емкость буфера совпадает с его пределом, поэтому метод `flip()` можно использовать вместо метода `rewind()`. Но эти два метода взаимозаменяемы далеко не всегда.

Как правило, буфер следует обнулять между любыми операциями чтения и записи. Например, в результате выполнения приведенного ниже цикла, составленного на основе предыдущего примера, английский алфавит будет записан в файл три раза. Обратите особое внимание на то, что метод `rewind()` вызывается каждый раз в промежутке между операциям чтения и записи.

```

int h=0; h<3; h++) {
    // записать заданное количество байтов в буфер
    for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));
}

```

```

// подготовить буфер к записи данных
mBuf.rewind();

// записать данные из буфера в выходной файл
fChan.write(mBuf);

// снова подготовить буфер к записи данных
mBuf.rewind();
}

```

В отношении рассматриваемой здесь программы следует также иметь в виду, что в процессе записи данных из буфера в файл первые 26 байт в файле будут содержать выводимые данные. Если файл `test.txt` существовал ранее, то после выполнения программы первые 26 байт в файле `test.txt` будут содержать алфавит, а остальная часть файла останется без изменения.

Еще один способ записи данных в файл подразумевает его сопоставление с буфером. Преимущество такого подхода заключается в том, что занесенные в буфер данные будут автоматически записаны в файл. Никаких явных операций записи не требуется. Для сопоставления и записи содержимого буфера в файла необходимо придерживаться следующей общей процедуры. Сначала получается объект типа `Path`, инкапсулирующий файл, а затем создается канал к этому файлу, для чего вызывается метод `Files.newByteChannel()`, которому передается объект типа `Path`. Ссылку, возвращаемую методом `newByteChannel()`, следует привести к типу `FileChannel`. Затем для канала вызывается метод `map()`, чтобы сопоставить канал с буфером. Метод `map()` был подробно описан в предыдущем разделе, а здесь он упоминается ради удобства изложения. Ниже приведена его общая форма.

```

MappedByteBuffer map(
    FileChannel.MapMode способ, long позиция, long размер)
    throws IOException

```

Метод `map()` сопоставляет данные из файла с буфером в памяти. Значение параметра *способ* определяет разрешенные операции. Чтобы записать данные в файл, в качестве параметра *способ* следует указать значение `MapMode.READ_WRITE`. Место для начала сопоставления в файле определяется параметром *позиция*, а количество сопоставляемых байтов — параметром *размер*. В итоге возвращается ссылка на буфер. Как только файл будет сопоставлен с буфером, в буфер можно вывести данные, которые будут автоматически записываться в файл. Поэтому никаких явных операций записи в канал не требуется.

Ниже приведена новая версия программы из предыдущего примера, переделанная таким образом, чтобы использовать сопоставление для записи данных в файл. Обратите внимание на то, что при вызове метода `newByteChannel()` в качестве параметра указывается значение `StandardOpenOption.READ`. Дело в том, что сопоставляемый буфер может использоваться только для чтения или же для чтения и записи. Таким образом, для записи в сопоставляемый буфер канал должен быть открыт как для чтения, так и для записи.

```

// Записать данные в сопоставляемый файл
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;

```

```

import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelWrite {
    public static void main(String args[]) {

        // получить канал к файлу в блоке try с ресурсами
        try ( FileChannel fChan = (FileChannel)
            Files.newByteChannel(Paths.get("test.txt"),
                StandardOpenOption.WRITE,
                StandardOpenOption.READ,
                StandardOpenOption.CREATE) )
        {

            // затем сопоставить файл с буфером
            MappedByteBuffer mBuf =
                fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // записать заданное количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte) ('A' + i));
        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}

```

Как видите, в данном примере отсутствуют явные операции записи непосредственно в канал. Буфер `mBuf` сопоставляется с файлом, поэтому изменения в буфере автоматически отражаются в основном файле.

## Копирование файлов средствами системы ввода-вывода NIO

Система ввода-вывода NIO упрощает несколько видов файловых операций. Хотя здесь недостаточно места, чтобы рассмотреть все эти операции, приведенный ниже пример программы дает общее представление о доступных средствах. В этой программе файл копируется единственным методом `copy()` — статическим методом из класса `Files` в системе ввода-вывода NIO. У этого метода имеется несколько общих форм. Ниже приведена та общая форма, которая будет использоваться в примерах, представленных далее.

```

static Path copy(Path источник, Path адресат, CopyOption ... способ)
throws IOException

```

Файл, определяемый параметром *источник*, копируется в файл, обозначаемый параметром *адресат*. А порядок копирования определяется параметром *способ*. Это параметр переменной длины, поэтому он может отсутствовать. Если же он определен, то позволяет передать одно или несколько приведенных ниже значений, допустимых для всех файловых систем. В зависимости от конкретной реализации могут поддерживаться и другие значения.

<code>StandardCopyOption.COPY_ATTRIBUTES</code>	Запросить копирование атрибутов файла
<code>StandardLinkOption.NOFOLLOW_LINKS</code>	Не следовать по символическим ссылкам
<code>StandardCopyOption.REPLACE_EXISTING</code>	Перезаписать прежний файл

В приведенном ниже примере программы демонстрируется применение метода `copy()`. Исходный и результирующие файлы указываются в командной строке, причем исходный файл указывается первым. Обратите внимание на краткость программы. Если сравнить эту версию программы копирования файла с ее аналогом из главы 13, то окажется, что та часть программы, которая фактически копирует файл, существенно короче в представленной здесь версии на основе системы ввода-вывода NIO.

```
// Скопировать файл средствами системы ввода-вывода NIO.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {

    public static void main(String args[]) {

        if(args.length != 2) {
            System.out.println("Применение: откуда и куда копировать");
            return;
        }

        try {
            Path source = Paths.get(args[0]);
            Path target = Paths.get(args[1]);

            // скопировать файл
            Files.copy(source, target,
                StandardCopyOption.REPLACE_EXISTING);

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

## Применение системы NIO для потокового ввода-вывода

Начиная с версии NIO.2, систему NIO можно использовать для открытия потока ввода-вывода. Получив объект типа `Path`, следует открыть файл, вызвав статический метод `newInputStream()` или `newOutputStream()`, определенный в классе `Files`. Эти методы возвращают поток ввода-вывода, связанный с указанным файлом. В любом случае поток ввода-вывода может быть затем использован так, как описано в главе 20, и для этого пригодны те же самые способы. Преимущество использования объекта типа `Path` для открытия файла заключается в том, что доступны все средства системы ввода-вывода NIO.

Для открытия файла с целью потокового ввода служит метод `Files.newInputStream()`. Он имеет следующую общую форму:

```
static InputStream newInputStream(Path путь, OpenOption ... способ)
    throws IOException
```

где параметр *путь* обозначает открываемый файл, а параметр *способ* — порядок открытия файла. Это параметр переменной длины, и поэтому он должен принимать одно или несколько значений, определенных в упомянутом ранее классе `StandardOpenOption`. (Безусловно, в данном случае применимы только те значения, которые относятся к потоку ввода.) Если же параметр *способ* не определен, то файл открывается так, как будто в качестве этого параметра передано значение `StandardOpenOption.READ`. После открытия файла можно использовать любой из методов, определенных в классе `InputStream`. Например, метод `read()` можно использовать для чтения байтов из файла.

В приведенном ниже примере программы демонстрируется применение потокового ввода-вывода на основе системы NIO. Этот пример содержит версию программы `ShowFile` из главы 13, переделанную таким образом, чтобы для открытия файла и получения потока ввода-вывода использовались средства системы NIO. Нетрудно заметить, что эта версия программы очень похожа на первоначальный ее вариант, за исключением используемого интерфейса `Path` и метода `newInputStream()`.

```

/* Эта программа выводит текстовый файл, используя код
потокового ввода-вывода на основе системы NIO.
Требуется установка комплекта JDK, начиная с версии 7

Чтобы воспользоваться этой программой, укажите имя файла,
который требуется просмотреть. Например, чтобы просмотреть
файл TEST.TXT, введите в режиме командной строки
следующую команду:

java ShowFile TEST.TXT
*/

import java.io.*;
import java.nio.file.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;

        // сначала удостовериться, что указано имя файла
        if(args.length != 1) {
            System.out.println("Применение: ShowFile имя_файла");
            return;
        }

        // открыть файл и получить связанный с ним поток ввода-вывода
        try (InputStream fin = Files.newInputStream(Paths.get(args[0])))
        {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}

```

Поток ввода-вывода, возвращаемый методом `newInputStream()`, является обычным, и поэтому он применяется как и любой другой поток ввода-вывода. Например, один поток ввода можно заключить в оболочку другого, буферизованного потока ввода, например, типа `BufferedInputStream`, чтобы обеспечить буферизацию так, как показано ниже. В итоге все операции чтения будут автоматически буферизованы.

```
new BufferedInputStream(Files.newInputStream(Paths.get(args[0])))
```

Для открытия файла с целью вывода служит метод `Files.newOutputStream()`, который имеет следующую общую форму:

```
static OutputStream newOutputStream(Path путь, OpenOption ... способ)  
throws IOException
```

где параметр *путь* обозначает открываемый файл, а параметр *способ* — порядок открытия файла. Это параметр переменной длины, и поэтому он должен принимать одно или несколько значений, определенных в упомянутом ранее классе `StandardOpenOption`. (Безусловно, в данном случае применимы только те значения, которые относятся к потоку вывода.) Если же параметр *способ* не определен, то файл открывается так, как будто в качестве этого параметра переданы значения `StandardOpenOption.WRITE`, `StandardOpenOption.CREATE` и `StandardOpenOption.TRUNCATE_EXISTING`.

Метод `newOutputStream()` применяется таким же способом, как и описанный ранее метод `newInputStream()`. После открытия файла можно вызвать любой метод, определенный в классе `OutputStream`. Например, вызвать метод `write()` для записи байтов в файл. Кроме того, поток вывода можно заключить в поток вывода байтов типа `BufferedOutputStream`, чтобы буферизовать его.

В приведенном ниже примере программы демонстрируется применение метода `newOutputStream()`. В этой программе английский алфавит записывается в файл `test.txt`. Обратите внимание на использование буферизованного ввода-вывода.

```
// Продемонстрировать потоковый вывод на основе системы NIO  
// Требуется установка комплекта JDK, начиная с версии 7  
  
import java.io.*;  
import java.nio.file.*;  
  
class NIOStreamWrite {  
    public static void main(String args[])  
    {  
        // открыть файл и получить связанный с ним поток вывода  
        try (OutputStream fout =  
            new BufferedOutputStream(  
                Files.newOutputStream(Paths.get("test.txt"))) )  
        {  
            // вывести в поток заданное количество байтов  
            for(int i=0; i < 26; i++)  
                fout.write((byte) ('A' + i));  
        } catch(InvalidPathException e) {  
            System.out.println("Ошибка указания пути " + e);  
        } catch(IOException e) {  
            System.out.println("Ошибка ввода-вывода: " + e);  
        }  
    }  
}
```

## Применение системы ввода-вывода NIO для операций в файловой системе

В начале главы 20 был представлен класс `File`, входящий в пакет `java.io`. Как упоминалось ранее, класс `File` обращается к файловой системе и оперирует различными атрибутами файлов, обозначающими, например, доступ только для чтения, скрытый файл и т.д. Он служит также для получения сведений о пути к файлу. Начиная с версии JDK 7 интерфейсы и классы, определенные в системе ввода-вывода NIO.2, предоставляют лучший способ выполнения этих операций. К их преимуществам относятся улучшенная поддержка символических ссылок, обхода дерева каталогов, усовершенствованная обработка метаданных и многое другое. В последующих подразделах приведены примеры двух наиболее распространенных операций в файловой системе: получения сведений о пути к файлу и самом файле, а также сведений о содержимом каталога.

---

**Помните!** Если требуется обновить устаревший код, в котором применяется класс `java.io.File`, новым кодом, в котором применяется интерфейс `Path`, воспользуйтесь методом `toPath()`, чтобы получить экземпляр интерфейса `Path` из экземпляра класса `File`.

### Получение сведений о пути к файлу и самом файле

Сведения о пути к файлу могут быть получены методами, определенными в интерфейсе `Path`. Некоторые атрибуты файлов, описываемые в интерфейсе `Path` (например, скрытый файл), получаются методами, определенными в классе `Files`. В рассматриваемом здесь примере употребляются такие методы из интерфейса `Path`, как `getName()`, `getParent()` и `toAbsolutePath()`, а также методы `isExecutable()`, `isHidden()`, `isReadable()`, `isWritable()` и `exists()` из класса `Files` (они представлены в табл. 21.5 и 21.6).

---

**Внимание!** Такими методами, как `isExecutable()`, `isReadable()`, `isWritable()` и `exists()`, следует пользоваться осторожно, поскольку состояние файловой системы после их вызова может измениться, что может привести к нарушению нормальной работы программы и отрицательно сказаться на состоянии защиты системы.

Другие атрибуты файлов получают по запросу из списка, создаваемого при вызове метода `Files.readAttributes()`. В рассматриваемом здесь примере программы этот метод вызывается для получения связанного с файлом объекта типа `BasicFileAttributes`, но аналогичный общий подход можно применить и к другим типам атрибутов.

В приведенном ниже примере программы демонстрируется применение некоторых методов из интерфейса `Path` и класса `Files` наряду с методами из интерфейса `BasicFileAttributes`. В этой программе подразумевается, что файл `test.txt` находится в каталоге `examples`, входящем в текущий каталог.

```
// Получить сведения о пути к файлу и самом файле
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
```

```

import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
    public static void main(String args[]) {
        Path filepath = Paths.get("examples\\test.txt");

        System.out.println("Имя файла: " + filepath.getName(1));
        System.out.println("Путь к файлу: " + filepath);
        System.out.println("Абсолютный путь к файлу: " +
            filepath.toAbsolutePath());
        System.out.println(
            "Родительский каталог: " + filepath.getParent());
        if(Files.exists(filepath))
            System.out.println("Файл существует");
        else
            System.out.println("Файл не существует");

        try {
            if(Files.isHidden(filepath))
                System.out.println("Файл скрыт");
            else
                System.out.println("Файл не скрыт");
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }

        Files.isWritable(filepath);
        System.out.println("Файл доступен для записи");

        Files.isReadable(filepath);
        System.out.println("Файл доступен для чтения");

        try {
            BasicFileAttributes attribs =
                Files.readAttributes(filepath, BasicFileAttributes.class);

            if(attribs.isDirectory())
                System.out.println("Это каталог");
            else
                System.out.println("Это не каталог");

            if(attribs.isRegularFile())
                System.out.println("Это обычный файл");
            else
                System.out.println("Это не обычный файл");

            if(attribs.isSymbolicLink())
                System.out.println("Это символическая ссылка");
            else
                System.out.println("Это не символическая ссылка");

            System.out.println("Время последней модификации файла: " +
                attribs.lastModifiedTime());
            System.out.println("Размер файла: " + attribs.size() +
                " байтов");
        } catch(IOException e) {
            System.out.println("Ошибка чтения атрибутов: " + e);
        }
    }
}

```

Если запустить эту программу на выполнение из каталога `MyDir`, в котором имеется каталог `examples`, содержащий файл `test.txt`, то в конечном итоге будет выведен результат, аналогичный приведенному ниже. (Разумеется, размер файла и его временные характеристики будут иными.)

```
Имя файла: test.txt
Путь к файлу: examples\test.txt
Абсолютный путь к файлу: C:\MyDir\examples\test.txt
Родительский каталог: examples
Файл существует
Файл не скрыт
Файл доступен для записи
Файл доступен для чтения
Это не каталог
Это обычный файл
Это не символическая ссылка
Время последней модификации файла: 2014-01-01T18:20:46.380445Z
Размер файла: 18 байтов
```

Если вы пользуетесь компьютером с файловой системой FAT (т.е. файловой системой DOS), то попытайтесь применить методы, определенные в интерфейсе `DosFileAttributes`. А если вы пользуетесь системой, совместимой с POSIX, то попробуйте применить методы, определенные в интерфейсе `PosixFileAttributes`.

## Получение содержимого каталога

Если путь описывает каталог, можно прочесть содержимое этого каталога, используя статические методы, определенные в классе `Files`. Для этого следует сначала получить поток ввода из каталога, вызвав метод `newDirectoryStream()` и передав ему объект типа `Path`, обозначающий каталог. Ниже приведена одна из форм метода `newDirectoryStream()`.

```
static DirectoryStream<Path> newDirectoryStream(Path путь_к_каталогу)
    throws IOException
```

Здесь параметр `путь_к_каталогу` инкапсулирует путь к конкретному каталогу. Этот метод возвращает объект типа `DirectoryStream<Path>`, применяемый для получения содержимого каталога. Он генерирует исключение типа `IOException` при возникновении ошибки ввода-вывода, а также исключение типа `NotDirectoryException` (его класс является производным от класса `IOException`), если указанный путь не приводит к каталогу. Кроме того, может быть сгенерировано исключение типа `SecurityException`, если доступ к каталогу запрещен.

Класс `DirectoryStream<Path>` реализует интерфейс `AutoCloseable`, поэтому объектом этого класса можно управлять в блоке оператора `try` с ресурсами. Этот класс реализует также интерфейс `Iterable<Path>`. Это означает, что содержимое каталога можно получить, перебрав содержимое объекта типа `DirectoryStream`. При переборе каждая запись каталога представлена экземпляром интерфейса `Path`. Простейший способ перебрать объект типа `DirectoryStream` — организовать цикл `for` в стиле `for each`. Следует, однако, иметь в виду, что итератор, реализуемый в классе `DirectoryStream<Path>`, может быть получен только один раз для каждого экземпляра. Следовательно, метод `iterator()` может быть вызван, а цикл `for` в стиле `for each` — выполнен только один раз.

В следующем примере программы выводится содержимое каталога MyDir:

```
// Вывести содержимое каталога. Требуется установка комплекта JDK,
// начиная с версии 7

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        // получить и обработать поток ввода каталога
        // в блоке оператора try
        try ( DirectoryStream<Path> dirstrm =
            Files.newDirectoryStream(Paths.get(dirname)) )
        {
            System.out.println("Каталог " + dirname);

            // Класс DirectoryStream реализует интерфейс Iterable,
            // поэтому для вывода содержимого каталога можно
            // организовать цикл for в стиле for each
            for(Path entry : dirstrm) {
                BasicFileAttributes attrs =
                    Files.readAttributes(entry, BasicFileAttributes.class);

                if(attrs.isDirectory())
                    System.out.print("<DIR> ");
                else
                    System.out.print("      ");

                System.out.println(entry.getName(1));
            }
        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(NotDirectoryException e) {
            System.out.println(dirname + " не является каталогом.");
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Эта программа выводит следующий результат:

```
Каталог \MyDir
  DirList.class
  DirList.java
<DIR> examples
  Test.txt
```

Содержимое каталога можно отфильтровать двумя способами. Самый простой из них — воспользоваться следующей общей формой метода `newDirectoryStream()`:

```
static DirectoryStream<Path> newDirectoryStream(
    Path путь_к_каталогу, String шаблон)
    throws IOException
```

В этой форме получаются только те файлы, имена которых совпадают с заданным *шаблоном*. В качестве параметра *шаблон* можно указать полное имя файла или маску. *Маска* — это символьная строка, определяющая глобальный или общий шаблон, с которым будет совпадать один или несколько файлов, и содержащая общеупотребительные метасимволы \* и ?. Они соответствуют любому количеству символов и любому одиночному символу соответственно. Ниже приведены другие шаблоны для сопоставления с маской.

---

<b>**</b>	Совпадает с любым количеством различных символов в каталогах
<b>[символы]</b>	Совпадает с любым из указанных <i>символов</i> . Символы * и ? среди указанных <i>символов</i> будут рассматриваться как обычные символы, а не как метасимволы. Через дефис можно указать пределы сопоставления с шаблоном, например, <b>[x-z]</b>
<b>{список_масок}</b>	Совпадает с любой из масок, задаваемых в <i>списке_масок</i> через запятую

---

Метасимволы \* и ? можно указать, используя последовательности символов \\* и \?, а для того чтобы указать знак \ — последовательность символов \\. Можете поэкспериментировать с маской, подставив ее в вызов метода `newDirectoryStream()` из предыдущего примера программы следующим образом:

```
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```

Еще один способ отфильтровать каталог — воспользоваться приведенной ниже общей формой метода `newDirectoryStream()`.

```
static DirectoryStream<Path> newDirectoryStream(Path путь_к_каталогу,
        DirectoryStream.Filter<? super Path> фильтр_файлов)
        throws IOException
```

Здесь `DirectoryStream.Filter` — это интерфейс, в котором определяется следующий метод:

```
boolean accept(T элемент) throws IOException
```

В данном случае типом **T** будет `Path`. Если требуется включить указанный *элемент* в список, возвращается логическое значение `true`, а иначе — логическое значение `false`. Эта форма метода `newDirectoryStream()` предоставляет возможность отфильтровать каталог по другому критерию, кроме имени файла. В частности, каталог можно отфильтровать по размеру, дате создания, дате модификации или атрибуту.

Этот процесс демонстрируется в приведенном ниже примере программы. В этой программе перечисляются только те файлы, которые доступны для записи.

```
// Вывести только те файлы из каталога,
// которые доступны для записи

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        // создать фильтр, возвращающий логическое значение true
```

```

// только в отношении доступных для записи файлов
DirectoryStream.Filter<Path> how =
    new DirectoryStream.Filter<Path>() {
        public boolean accept(Path filename) throws IOException {
            if(Files.isWritable(filename)) return true;
            return false;
        }
    };

// получить и использовать поток ввода из каталога
// только доступных для записи файлов
try (DirectoryStream<Path> dirstrm =
    Files.newDirectoryStream(Paths.get(dirname), how) )
{
    System.out.println("Каталог " + dirname);

    for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
            Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
            System.out.print("<DIR> ");
        else
            System.out.print("      ");

        System.out.println(entry.getName(1));
    }
} catch (InvalidPathException e) {
    System.out.println("Ошибка указания пути " + e);
} catch (NotDirectoryException e) {
    System.out.println(dirname + " не является каталогом.");
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода: " + e);
}
}
}

```

## Обход дерева каталогов с помощью метода `walkFileTree()`

В предыдущих примерах получалось содержимое только одного каталога. Но иногда требуется получить список файлов из дерева каталогов. В прошлом решить подобную задачу было нелегко, но система ввода-вывода NIO.2 значительно упрощает ее решение, поскольку теперь из класса `Files` можно вызвать метод `walkFileTree()`, способный обработать дерево каталогов. Этот метод имеет две общие формы объявления. В примерах, приведенных в этой главе, употребляется следующая форма этого метода:

```

static Path walkFileTree(Path корень, FileVisitor<? extends Path> fv)
    throws IOException

```

Исходная точка обхода дерева каталогов передается в качестве параметра `корень`. Экземпляр интерфейса `FileVisitor` передается в качестве параметра `fv`. Реализация интерфейса `FileVisitor` определяет способ обхода дерева каталогов, а также позволяет обращаться к сведениям о каталоге. При возникновении ошибки ввода-вывода генерируется исключение типа `IOException`. Кроме того, может быть сгенерировано исключение типа `SecurityException`.

В интерфейсе `FileVisitor` определяется, каким образом посещаются файлы при обходе дерева каталогов. Этот обобщенный интерфейс объявляется следующим образом:

```
interface FileVisitor<T>
```

При вызове метода `walkFileTree()` в качестве параметра типа `T` указывается интерфейс `Path` (или любой производный от него тип). В интерфейсе `FileVisitor` определены методы, перечисленные в табл. 21.11.

**Таблица 21.11. Методы из интерфейса `FileVisitor`**

Метод	Описание
<code>FileVisitResult postVisitDirectory(T каталог, IOException исключение) throws IOException</code>	Вызывается после посещения каталога. Каталог передается в качестве параметра <i>каталог</i> , а любое исключение типа <code>IOException</code> — в качестве параметра <i>исключение</i> . Если параметр <i>исключение</i> принимает пустое значение <code>null</code> , то никакого исключения не происходит. Возвращает полученный результат
<code>FileVisitResult preVisitDirectory(T каталог, BasicFileAttributes атрибуты) throws IOException</code>	Вызывается перед посещением каталога. Каталог передается в качестве параметра <i>каталог</i> , а связанные с ним атрибуты — в качестве параметра <i>атрибуты</i> . Возвращает полученный результат. Чтобы исследовать каталог, следует вернуть значение <code>FileVisitResult.CONTINUE</code>
<code>FileVisitResult visitFile(T файл, BasicFileAttributes атрибуты) throws IOException</code>	Вызывается при посещении файла. Файл передается в качестве параметра <i>файл</i> , а связанные с ним атрибуты — в качестве параметра <i>атрибуты</i> . Возвращает полученный результат
<code>FileVisitResult visitFileFailed(T файл, IOException исключение) throws IOException</code>	Вызывается при неудачной попытке посетить файл. Файл, который не удалось посетить, передается в качестве параметра <i>файл</i> , а исключение типа <code>IOException</code> — в качестве параметра <i>исключение</i> . Возвращает полученный результат

Обратите внимание на то, что каждый метод возвращает значение из перечисления `FileVisitResult`. В этом перечислении определяются следующие значения:

<code>CONTINUE</code>	<code>SKIP_SIBLINGS</code>	<code>SKIP_SUBTREE</code>	<code>TERMINATE</code>
-----------------------	----------------------------	---------------------------	------------------------

В общем, для продолжения обхода каталога и находящихся в нем каталогов метод должен вернуть значение `CONTINUE`. Для того чтобы пропустить каталог и его содержимое, а также предотвратить вызов метода `postVisitDirectory()`, из метода `preVisitDirectory()` должно быть возвращено значение `SKIP_SIBLINGS`, чтобы пропустить только каталог и подкаталоги — значение `SKIP_SUBTREE`, а для того чтобы остановить обход каталога — значение `TERMINATE`.

Безусловно, можно создать собственный класс для обхода каталогов и реализовать в нем методы, определенные в интерфейсе `FileVisitor`, но обычно так не поступают, поскольку предоставляется простая их реализация в классе `SimpleFileVisitor`. В этом случае достаточно переопределить реализацию

по умолчанию одного или нескольких нужных методов. Ниже приведен краткий пример программы, демонстрирующий этот процесс. В этой программе выводятся все файлы из дерева каталогов, в корне которого находится каталог `\MyDir`. Обратите внимание на краткость этой программы.

```
// Простой пример применения метода walkFileTree()
// для вывода дерева каталогов. Требуется установка
// комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

// создать специальную версию класса SimpleFileVisitor,
// в которой переопределяется метод visitFile()
class MyFileVisitor extends SimpleFileVisitor<Path> {
    public FileVisitResult visitFile(
        Path path, BasicFileAttributes attribs)
        throws IOException
    {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
}

class DirTreeList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        System.out.println("Дерево каталогов, начиная с каталога " +
            dirname + ":\n");
        try {
            Files.walkFileTree(Paths.get(dirname), new MyFileVisitor());
        } catch (IOException exc) {
            System.out.println("Ошибка ввода-вывода");
        }
    }
}
```

Ниже приведен примерный результат, выводимый данной программой для обхода того же каталога `MyDir`, что и прежде. В данном примере подкаталог `examples` содержит только один файл `MyProgram.java`.

Дерево каталогов, начиная с каталога `\MyDir`:

```
\MyDir\DirList.class
\MMyDir\DirList.java
\MMyDir\examples\MyProgram.java
\MMyDir\Test.txt
```

В данной программе класс `MyFileVisitor` расширяет класс `SimpleFileVisitor`, переопределяя только метод `visitFile()`, который просто выводит их, хотя совсем не трудно достичь и более сложных функциональных возможностей. Например, можно было бы отфильтровать файлы или выполнить над ними такие действия, как копирование на резервное устройство. Ради простоты в данном примере для переопределения метода `visitFile()` выбран именованный класс, но вместо него ничто не мешает воспользоваться анонимным внутренним классом. И последнее замечание: средствами класса `java.nio.file.WatchService` можно отследить изменения в каталоге.

## Примеры организации канального ввода-вывода до версии JDK 7

Прежде чем завершить эту главу, следует рассмотреть еще одну особенность системы ввода-вывода NIO. В приведенных выше примерах употреблялись некоторые из новых средств, внедренных в систему ввода-вывода NIO, начиная с версии JDK 7. Но ведь имеется еще немало кода, который написан до версии JDK 7 и требует надлежащего сопровождения, а возможно, и преобразования для применения новых средств. Поэтому в последующих разделах будет показано, как организовать чтение и запись в файлы средствами системы ввода-вывода NIO, доступными до версии JDK 7. Некоторые из приведенных выше примеров переделаны таким образом, чтобы использовать предыдущие средства системы ввода-вывода NIO, а не новые средства, поддерживаемые в системе ввода-вывода NIO.2. Это означает, что приведенные далее примеры пригодны для версий Java, выпущенных до версии JDK 7.

Главное отличие прежнего кода от нового, в котором применяется система ввода-вывода NIO, заключается в интерфейсе `Path`, который был внедрен в версии JDK 7. Следовательно, для описания файла или открытия канала к нему в прежнем коде не применяется интерфейс `Path`. Кроме того, в прежнем коде не применяются операторы `try` с ресурсами, поскольку автоматическое управление ресурсами также было внедрено только в версии JDK 7.

---

**Помните!** В примерах программ из этого раздела описывается действие унаследованного кода, в котором применяется система ввода-вывода NIO. Материал этого раздела предназначен для тех программистов, которые продолжают работать с унаследованным кодом или пользуются компилятором, выпущенным до версии JDK 7. В новом коде должны применяться средства системы ввода-вывода NIO, внедренные в версии JDK 7.

## Чтение из файла до версии JDK 7

В этом разделе рассматриваются два предыдущих примера канального ввода из файла, переделанных для использования средств системы ввода-вывода NIO, доступных только до версии JDK 7. В первом примере для чтения данных из файла сначала выделяется вручную оперативная память под буфер, а затем выполняется явным образом операция чтения. Во втором примере производится сопоставление файла с буфером, автоматизирующее весь процесс чтения данных из файла.

Если для чтения данных из файла через канал и выделяемый вручную буфер используются версии Java, выпущенные до версии JDK 7, то сначала файл открывается с помощью потока ввода типа `FileInputStream` таким же способом, как описано в главе 20. Затем вызывается метод `getChannel()` для потока ввода типа `FileInputStream`, чтобы получить канал к открытому файлу. Ниже приведена общая форма этого метода.

```
FileChannel getChannel()
```

В этой форме возвращается объект типа `FileChannel`, инкапсулирующий канал для файловых операций. Далее вызывается метод `allocate()` для выделения оперативной памяти под буфер. Файловые каналы оперируют буферами байтов, и поэтому для их получения в примерах этой главы вызывается метод `allocate()`, определенный в классе `ByteBuffer`, как пояснялось ранее.

В следующем примере программы демонстрируется чтение и вывод содержимого файла `test.txt` через канал с использованием явных операций ввода для версий Java, выпущенных до JDK 7:

```
// Использовать каналы для чтения данных из файла.
// Версия до JDK 7
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;
        int count;

        try {
            // сначала открыть файл для ввода
            fIn = new FileInputStream("test.txt");

            // затем получить канал к этому файлу
            fChan = fIn.getChannel();

            // выделить оперативную память под буфер
            mBuf = ByteBuffer.allocate(128);

            do {
                // читать данные в буфер
                count = fChan.read(mBuf);

                // прекратить чтение по достижении конца файла
                if(count != -1) {

                    // подготовить буфер к чтению из него данных
                    mBuf.rewind();

                    // читать байты данных из буфера и
                    // выводить их на экран как символы
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);

            System.out.println();

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Ошибка закрытия канала.");
            }
        }
    }
}
```



```
FileChannel fChan = null;
long fSize;
MappedByteBuffer mBuf;

try {
    // сначала открыть файл для ввода
    fIn = new FileInputStream("test.txt");

    // затем получить канал к этому файлу
    fChan = fIn.getChannel();

    // получить размер файла
    fSize = fChan.size();

    // а теперь сопоставить файл с буфером
    mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

    // читать байты из буфера и выводить их на экран
    for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода " + e);
} finally {
    try {
        try {
            if(fChan != null) fChan.close(); // закрыть канал
        } catch(IOException e) {
            System.out.println("Ошибка закрытия канала.");
        }
        try {
            if(fIn != null) fIn.close(); // закрыть файл
        } catch(IOException e) {
            System.out.println("Ошибка закрытия файла.");
        }
    }
}
}
```

В данной программе файл открывается с помощью конструктора класса `FileInputStream`, а ссылка на создаваемый объект этого класса присваивается переменной экземпляра `fIn`. Затем вызывается метод `getChannel()` по ссылке на этот объект в переменной экземпляра `fIn`, чтобы создать канал, подключаемый к открытому файлу. Далее определяется размер файла. После этого вызывается метод `map()`, чтобы сопоставить весь файл с областью в памяти, выделяемой под буфер, а ссылка на этот буфер сохраняется в переменной экземпляра `mBuf`. И наконец, вызывается метод `get()`, чтобы прочитать байты из буфера `mBuf`.

## Запись в файл до версии JDK 7

В этом разделе представлены два предыдущих примера канального вывода данных в файл, переделанные таким образом, чтобы использовать только средства системы ввода-вывода NIO, доступные до версии JDK 7. В первом примере для записи данных в файл сначала выделяется вручную оперативная память под буфер, а затем выполняется явным образом операция записи. А во втором примере производится сопоставление файла с буфером, автоматизирующее весь

процесс записи данных в файл. Но в обоих случаях ни интерфейс `Path`, ни оператор `try` с ресурсами не применяются, поскольку они доступны лишь с версии JDK 7.

Если для записи данных в файл через канал и выделяемый вручную буфер используются версии Java, выпущенные до версии JDK 7, то сначала файл открывается с помощью потока вывода типа `FileOutputStream` таким же способом, как описано в главе 20. Затем вызывается метод `getChannel()` для потока вывода типа `FileOutputStream`, чтобы получить канал к открытому файлу, а после этого — метод `allocate()`, чтобы выделить оперативную память под буфер, как описано в предыдущем разделе. Далее записываемые в файл данные размещаются в этом буфере и вызывается метод `write()` для открытого канала. Вся эта процедура демонстрируется в приведенном ниже примере программы. В этой программе английский алфавит записывается в файл `test.txt`.

```
// Записать данные в файл средствами системы ввода-вывода NIO.
// Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
    public static void main(String args[]) {
        FileOutputStream fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            // сначала открыть файл для вывода данных
            fOut = new FileOutputStream("test.txt");

            // затем получить канал к файлу для вывода данных
            fChan = fOut.getChannel();

            // создать буфер
            mBuf = ByteBuffer.allocate(26);

            // записать некоторое количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

            // подготовить буфер к записи данных
            mBuf.rewind();

            // записать данные из буфера в выходной файл
            fChan.write(mBuf);

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Ошибка закрытия канала.");
            }
            try {
                if(fOut != null) fOut.close(); // закрыть файл
            }
        }
    }
}
```

```

    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла.");
    }
}
}
}

```

Вызов метода `rewind()` по ссылке на буфер в переменной экземпляра `mBuf` требуется для возврата текущей позиции в начало буфера `mBuf` после записи данных. Напомним, что при каждом вызове метода `put()` текущая позиция продвигается к концу буфера. Поэтому, прежде чем вызвать метод `write()`, следует установить текущую позицию в начало буфера. В противном случае метод `write()` не сумеет обнаружить в буфере никаких данных, посчитав, что их там вообще нет.

Если для записи данных в файл производится сопоставление файла с буфером и для этой цели используются версии Java, выпущенные до версии JDK 7, то сначала для выполнения операций чтения и записи создается объект класса `RandomAccessFile`, чтобы открыть файл. Для открываемого файла требуется разрешение на чтение и запись. Затем для данного объекта вызывается метод `map()`, чтобы сопоставить открытый файл с буфером. Далее записываемые данные размещаются в буфере. А поскольку буфер сопоставлен с файлом, то любые изменения в нем автоматически отражаются в файле. Таким образом, никаких явных операций записи в канал не требуется.

Ниже приведена версия программы из предыдущего примера, переделанная для сопоставления файла с буфером.

```

// Записать данные в сопоставленный файл. Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
    public static void main(String args[]) {
        RandomAccessFile fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            fOut = new RandomAccessFile("test.txt", "rw");

            // получить канал к открытому файлу
            fChan = fOut.getChannel();

            // затем сопоставить файл с буфером
            mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // записать некоторое количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {

```

```
        System.out.println("Ошибка закрытия канала.");
    }
    try {
        if(fOut != null) fOut.close(); // закрыть файл
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла.");
    }
}
}
```

Как видите, в данном примере нет никаких явных операций записи непосредственно в канал. Благодаря тому что буфер `mBuf` сопоставляется с файлом, изменения в буфере автоматически отражаются в базовом файле.