

ГЛАВА

18

Пакет `java.util`, часть I. Collections Framework

С этой главы начинается рассмотрение классов и интерфейсов, определенных в пакете `java.util`. Этот важный пакет предлагает большой выбор классов и интерфейсов, поддерживающих обширный ряд функциональных возможностей. В частности, в пакет `java.util` входят классы, создающие псевдослучайные числа, управляющие датой и временем, просмотром событий, манипулирующие наборами битов, выполняющие синтаксический анализ символьных строк и обрабатывающие форматированные данные. В пакет `java.util` входит также одна из самых эффективных подсистем Java — каркас коллекций Collections Framework. Этот каркас представляет собой сложную иерархию интерфейсов и классов, реализующих современную технологию управления группами объектов. Он заслуживает пристального внимания всех программирующих на Java.

Пакет `java.util` предоставляет обширный ряд функциональных возможностей, поэтому он достаточно объемный. Ниже приведен перечень его основных классов.

<code>AbstractCollection</code>	<code>FormattableFlags</code>	<code>Properties</code>
<code>AbstractList</code>	<code>Formatter</code>	<code>PropertyPermission</code>
<code>AbstractMap</code>	<code>GregorianCalendar</code>	<code>PropertyResourceBundle</code>
<code>AbstractQueue</code>	<code>HashMap</code>	<code>Random</code>
<code>AbstractSequentialList</code>	<code>HashSet</code>	<code>ResourceBundle</code>
<code>AbstractSet</code>	<code>Hashtable</code>	<code>Scanner</code>
<code>ArrayDeque</code>	<code>IdentityHashMap</code>	<code>ServiceLoader</code>
<code>ArrayList</code>	<code>IntSummaryStatistics</code> (добавлен в версии JDK 8)	<code>SimpleTimeZone</code>
<code>Arrays</code>	<code>LinkedHashMap</code>	<code>Spliterators</code> (добавлен в версии JDK 8)
<code>Base64</code> (добавлен в версии JDK 8)	<code>LinkedHashSet</code>	<code>SplittableRandom</code> (добавлен в версии JDK 8)
<code>BitSet</code>	<code>LinkedList</code>	<code>Stack</code>
<code>Calendar</code>	<code>ListResourceBundle</code>	<code>StringJoiner</code> (добавлен в версии JDK 8)
<code>Collections</code>	<code>Locale</code>	<code>StringTokenizer</code>
<code>Currency</code>	<code>LongSummaryStatistics</code> (добавлен в версии JDK 8)	<code>Timer</code>
<code>Date</code>	<code>Objects</code>	<code>TimerTask</code>
<code>Dictionary</code>	<code>Observable</code>	<code>TimeZone</code>

DoubleSummaryStatistics (добавлен в версии JDK 8)	Optional (добавлен в версии JDK 8)	TreeMap
EnumMap	OptionalDouble (добавлен в версии JDK 8)	TreeSet
EnumSet	OptionalInt (добавлен в версии JDK 8)	UUID
EventListenerProxy	OptionalLong (добавлен в версии JDK 8)	Vector
EventObject	PriorityQueue	WeakHashMap

В пакете `java.util` определены следующие интерфейсы:

Collection	Map.Entry	Set
Comparator	NavigableMap	SortedMap
Deque	NavigableSet	SortedSet
Enumeration	Observer	Spliterator (добавлен в версии JDK 8)
EventListener	PrimitiveIterator (добавлен в версии JDK 8)	Spliterator.OfDouble (добавлен в версии JDK 8)
Formattable	PrimitiveIterator.OfDouble (добавлен в версии JDK 8)	Spliterator.OfInt (добавлен в версии JDK 8)
Iterator	PrimitiveIterator.OfInt (добавлен в версии JDK 8)	Spliterator.OfLong (добавлен в версии JDK 8)
List	PrimitiveIterator.OfLong (добавлен в версии JDK 8)	Spliterator.OfPrimitive (добавлен в версии JDK 8)
ListIterator	Queue	
Map	RandomAccess	

Из-за большого размера пакета `java.util` его описание разделено на две главы. Эта глава посвящена средствам из каркаса коллекций Collections Framework, входящего в пакет `java.util`. А в главе 19 рассматриваются остальные классы и интерфейсы из этого пакета.

Краткий обзор коллекций

Каркас коллекций Collections Framework в Java стандартизирует способы управления группами объектов в прикладных программах. Коллекции не были частью исходной версии языка Java, но были внедрены в версии J2SE 1.2. До появления каркаса коллекций для хранения групп объектов и манипулирования ими в Java предоставлялись такие специальные классы, как `Dictionary`, `Vector`, `Stack` и `Properties`. И хотя эти классы были достаточно удобны, им недоставало общей, объединяющей основы. Так, класс `Vector` отличался способом своего применения от класса `Properties`. Такой первоначальный специализированный подход не был рассчитан на дальнейшее расширение и адаптацию. Для разрешения этого и ряда других затруднений и были внедрены коллекции.

Каркас коллекций был разработан для достижения нескольких целей. Во-первых, он должен был обеспечивать высокую производительность. Реализация основных коллекций (динамических массивов, связных списков, деревьев и хеш-

таблиц) отличается высокой эффективностью. Программировать один из таких “механизмов доступа к данным” вручную приходится крайне редко. Во-вторых, каркас должен был обеспечивать единообразное функционирование коллекций с высокой степенью взаимодействия. В-третьих, коллекции должны были допускать простое расширение и/или адаптацию. В этом отношении весь каркас коллекций построен на едином наборе стандартных интерфейсов. Некоторые стандартные реализации этих интерфейсов (например, в классах `LinkedList`, `HashSet` и `TreeSet`) можно использовать в исходном виде. Но при желании можно реализовать и свои коллекции. Для удобства программистов предусмотрены различные реализации специального назначения, а также частичные реализации, которые облегчают создание собственных коллекций. И наконец, в каркас коллекций были внедрены механизмы интеграции стандартных массивов.

Алгоритмы составляют другую важную часть каркаса коллекций. Алгоритмы оперируют коллекциями и определены в виде статических методов в классе `Collections`. Таким образом, они доступны всем коллекциям и не требуют реализаций их собственной версии в каждом классе коллекции. Алгоритмы предоставляют стандартные средства для манипулирования коллекциями.

Другим элементом, тесно связанным с каркасом коллекций, является интерфейс `Iterator`, определяющий *итератор*, который обеспечивает общий, стандартизованный способ поочередного доступа к элементам коллекций. Иными словами, итератор предоставляет способ *перебора содержимого коллекций*. А поскольку каждая коллекция предоставляет свой итератор, то элементы любого класса коллекций могут быть доступны с помощью методов, определенных в интерфейсе `Iterator`. Таким образом, код, перебирающий в цикле элементы множества, можно с минимальными изменениями применить, например, для перебора элементов списка.

В версии JDK 8 внедрена другая разновидность итератора, называемая *итератором-разделителем*. Если говорить коротко, то итераторы-разделители обеспечивают параллельную итерацию. Итераторы-разделители поддерживаются в интерфейсе `Spliterator` и ряде вложенных в него интерфейсов, которые, в свою очередь, поддерживают примитивные типы данных. В версии JDK 8 внедрены также интерфейсы итераторов, предназначенные для применения вместе с примитивными типами данных. К их числу относятся интерфейсы `PrimitiveIterator` и `PrimitiveIterator.OfDouble`.

Помимо коллекций, в каркасе Collections Framework определен ряд интерфейсов и классов *отображений*, в которых хранятся пары “ключ–значение”. Несмотря на то что отображения входят в состав каркаса коллекций, строго говоря, они не являются коллекциями. Тем не менее для отображения можно получить *представление коллекции*. Такое представление содержит элементы отображения, хранящиеся в коллекции. Таким образом, содержимое отображения можно при желании обрабатывать как коллекцию.

Механизм коллекций был усовершенствован для некоторых классов, изначально определенных в пакете `java.util` таким образом, чтобы интегрировать их в новую систему. Но несмотря на то что внедрение коллекций изменило архитектуру многих первоначальных служебных классов, они не стали от этого нерекомендованными к употреблению. Коллекции просто предлагают лучшее решение некоторых задач.

На заметку! Если у вас имеется некоторый опыт программирования на C++, то вам может помочь сходство коллекций в Java со стандартной библиотекой шаблонов (STL), определенной в C++. То, что в C++ называется **контейнером**, в Java именуется **коллекцией**. Но у каркаса коллекций Collections Framework и библиотеки STL имеются существенные отличия. Поэтому не делайте поспешных выводов.

Изменения каркаса коллекций в версии JDK 5

С выходом версии JDK 5 в каркасе коллекций Collections Framework произошел ряд существенный изменений, значительно повысивших его эффективность и упростивших его применение. К числу этих изменений относится внедрение обобщений, автоматическая упаковка и распаковка, а также организация цикла `for` в стиле `for each`. И хотя JDK 8 является уже третьей основной версией Java, появившейся после выпуска JDK 5, последствия изменений, внесенных в каркас коллекций в версии JDK 5, настолько значительны, что они до сих пор заслуживают особого внимания. Основная причина состоит в том, что по-прежнему существует и эксплуатируется большой объем кода, написанного до версии JDK 5. Понимать последствия и причины этих изменений очень важно на тот случай, если придется сопровождать или обновлять устаревший код.

Обобщения коренным образом изменили каркас коллекций

Внедрение обобщений коренным образом изменило каркас коллекций, поскольку он был полностью переделан в связи этим нововведением. Все коллекции теперь являются обобщенными, и многие методы, оперирующие коллекциями, также принимают обобщенные параметры. Проще говоря, внедрение обобщений коснулось всех частей каркаса коллекций.

Обобщения внесли в коллекции именно то, чего им явно недоставало: типовую безопасность. Раньше во всех коллекциях хранились ссылки на класс `Object`, а это означало, что в любой коллекции могли храниться объекты любого типа. Таким образом, в одной коллекции можно было непреднамеренно сохранить несовместимые типы данных. А это могло привести к ошибкам несовместимости типов во время выполнения. Благодаря обобщениям теперь можно явным образом указать тип сохраняемых данных и тем самым избежать подобных ошибок во время выполнения.

Несмотря на то что внедрение обобщений изменило объявления большинства классов и интерфейсов, а также некоторых их методов, в целом каркас коллекций действует таким же образом, как и до появления обобщений. Безусловно, чтобы извлечь выгоду, которую обобщения приносят коллекциям, придется переписать устаревший код. Это важно сделать еще и потому, что при компилировании кода, написанного до появления обобщений, современный компилятор будет выдавать предупреждающие сообщения. Во избежание подобных сообщений в код всех создаваемых коллекций придется ввести сведения об их типе.

В средствах автоматической упаковки используются примитивные типы данных

Автоматическая упаковка и распаковка упрощают сохранение данных примитивных типов в коллекциях. Как будет показано далее, в коллекциях можно сохранять только ссылки, но не значения примитивных типов. Если раньше требовалось сохранить в коллекции значение примитивного типа вроде `int`, то его приходилось вручную упаковывать в оболочку данного типа. А когда значение извлекалось из оболочки, нужно было его вручную распаковывать, приводя его явным образом к соответствующему примитивному типу. Упаковка и распаковка осуществляются теперь в Java автоматически, когда требуется сохранять или извлекать данные примитивных типов. Таким образом, выполнять эти операции вручную больше не нужно.

Цикл `for` в стиле `for each`

Все классы в каркасе коллекций усовершенствованы таким образом, чтобы реализовывать интерфейс `Iterable`. Это означает, что содержимое коллекции можно перебрать, организовав цикл `for` в стиле `for each`. Раньше для перебора содержимого коллекции нужно было использовать итератор, рассматриваемый далее в этой главе, организуя цикл вручную. Несмотря на то что итераторы по-прежнему применяются для некоторых целей, во многих случаях циклы на основе итераторов могут быть заменены циклами `for` в стиле `for each`.

Интерфейсы коллекций

В каркасе коллекций определяется несколько интерфейсов. В этом разделе дается краткий обзор каждого из них. Начать рассмотрение коллекций с их интерфейсов следует потому, что они определяют саму сущность классов коллекций. А конкретные классы лишь предоставляют различные реализации стандартных интерфейсов. Интерфейсы, поддерживающие коллекции, перечислены в табл. 18.1.

Таблица 18.1. Интерфейсы, поддерживающие коллекции

Интерфейс	Описание
<code>Collection</code>	Позволяет работать с группами объектов. Находится на вершине иерархии коллекций
<code>Deque</code>	Расширяет интерфейс <code>Queue</code> для организации двусторонних очередей
<code>List</code>	Расширяет интерфейс <code>Collection</code> для управления последовательностями (списками объектов)
<code>NavigableSet</code>	Расширяет интерфейс <code>SortedSet</code> для извлечения элементов по результатам поиска ближайшего совпадения
<code>Queue</code>	Расширяет интерфейс <code>Collection</code> для управления специальными типами списков, где элементы удаляются только из начала списка
<code>Set</code>	Расширяет интерфейс <code>Collection</code> для управления множествами, которые должны содержать однозначные элементы
<code>SortedSet</code>	Расширяет интерфейс <code>Set</code> для управления отсортированными множествами

Помимо перечисленных выше интерфейсов, для составления коллекций используются интерфейсы Comparator, RandomAccess, Iterator и ListIterator, которые подробнее рассматриваются далее в этой главе. А начиная с версии JDK 8 к их числу принадлежит также интерфейс Spliterator. Если говорить кратко, то интерфейс Comparator определяет два сравниваемых объекта, а интерфейсы Iterator, ListIterator и Spliterator перечисляют объекты в коллекции. Если же список реализует интерфейс RandomAccess, то тем самым он поддерживает эффективный произвольный доступ к своим элементам.

Ради обеспечения максимальных удобств применения интерфейсов коллекций некоторые методы в них могут быть необязательными. Необязательные методы позволяют видоизменять содержимое коллекций. Коллекции, поддерживающие такие методы, называются *изменяемыми*, а коллекции, не позволяющие изменять свое содержимое, — *неизменяемыми*. Если предпринимается попытка вызвать один из этих методов для неизменяемой коллекции, то генерируется исключение типа UnsupportedOperationException. Все встроенные коллекции являются изменяемыми. В последующих разделах подробно рассматриваются интерфейсы коллекций.

Интерфейс Collection

Этот интерфейс служит основанием, на котором построен весь каркас коллекций, поскольку он должен быть реализован всеми классами коллекций. Интерфейс Collection является обобщенным и объявляется следующим образом:

```
interface Collection<E>
```

где E обозначает тип объектов, которые будет содержать коллекция. Интерфейс Collection расширяет интерфейс Iterable. Это означает, что все коллекции можно перебирать, организовав цикл for в стиле for each. (Напомним, что только те классы, которые реализуют интерфейс Iterable, позволяют перебирать их элементы в цикле for.)

В интерфейсе Collection определяются основные методы, которые должны иметь все коллекции. Эти методы перечислены в табл. 18.2. В связи с тем что все коллекции реализуют интерфейс Collection, знакомство с его методами требуется для ясного понимания каркаса коллекций. Некоторые из этих методов могут генерировать исключение типа UnsupportedOperationException. Как пояснялось ранее, это исключение возникает в том случае, если коллекция не может быть изменена. Исключение типа ClassCastException генерируется в том случае, если объекты несовместимы, например, при попытке ввести несовместимый объект в коллекцию. Исключение типа NullPointerException генерируется при попытке ввести пустое значение null в коллекцию, не допускающую наличие пустых элементов. Исключение типа IllegalArgumentException генерируется в том случае, если указан неверный аргумент. А исключение типа IllegalStateException генерируется при попытке ввести новый элемент в заполненную коллекцию фиксированной длины.

Таблица 18.2. Методы из интерфейса `Collection`

Метод	Описание
<code>boolean add(E объект)</code>	Вводит заданный <i>объект</i> в вызывающую коллекцию. Возвращает логическое значение <code>true</code> , если <i>объект</i> успешно введен в коллекцию. А если <i>объект</i> уже присутствует в коллекции, которая не допускает дублирование объектов, то возвращается логическое значение <code>false</code>
<code>boolean addAll(Collection<? extends E> c)</code>	Вводит все элементы заданной коллекции <i>c</i> в вызывающую коллекцию. Возвращает логическое значение <code>true</code> , если коллекция изменена (т.е. все элементы введены), а иначе – логическое значение <code>false</code>
<code>void clear()</code>	Удаляет все элементы из вызывающей коллекции
<code>boolean contains(Object объект)</code>	Возвращает логическое значение <code>true</code> , если заданный <i>объект</i> является элементом вызывающей коллекции, а иначе – логическое значение <code>false</code>
<code>boolean containsAll(Collection<?> c)</code>	Возвращает логическое значение <code>true</code> , если вызывающая коллекция содержит все элементы заданной коллекции <i>c</i> , а иначе – логическое значение <code>false</code>
<code>boolean equals(Object объект)</code>	Возвращает логическое значение <code>true</code> , если вызывающая коллекция и заданный <i>объект</i> равнозначны, а иначе – логическое значение <code>false</code>
<code>int hashCode()</code>	Возвращает хеш-код вызывающей коллекции
<code>boolean isEmpty()</code>	Возвращает логическое значение <code>true</code> , если вызывающая коллекция пуста, а иначе – логическое значение <code>false</code>
<code>Iterator<E> iterator()</code>	Возвращает итератор для вызывающей коллекции
<code>default Stream<E> parallelStream()</code>	Возвращает поток, использующий вызывающую коллекцию в качестве источника для ввода-вывода элементов. В этом потоке поддерживаются параллельные операции ввода-вывода, если это вообще возможно (добавлен в версии JDK 8)
<code>boolean remove(Object объект)</code>	Удаляет один экземпляр <i>объекта</i> из вызывающей коллекции. Возвращает логическое значение <code>true</code> , если элемент удален, а иначе – логическое значение <code>false</code>
<code>boolean removeAll(Collection<?> c)</code>	Удаляет все элементы заданной коллекции <i>c</i> из вызывающей коллекции. Возвращает логическое значение <code>true</code> , если в конечном итоге коллекция изменяется (т.е. элементы из нее удалены), а иначе – логическое значение <code>false</code>
<code>default boolean removeIf(Predicate<? super E> предикат)</code>	Удаляет из вызывающей коллекции элементы, удовлетворяющие условию, которое задает <i>предикат</i> (добавлен в версии JDK 8)

Окончание табл. 18.2

Метод	Описание
<code>boolean retainAll(Collection<?> c)</code>	Удаляет из вызывающей коллекции все элементы, кроме элементов заданной коллекции <i>c</i> . Возвращает логическое значение <code>true</code> , если в конечном итоге коллекция изменяется (т.е. элементы из нее удалены), а иначе – логическое значение <code>false</code>
<code>int size()</code>	Возвращает количество элементов, содержащихся в коллекции
<code>default Spliterator<E> spliterator()</code>	Возвращает итератор-разделитель для вызывающей коллекции (добавлен в версии JDK 8)
<code>default Stream<E> stream()</code>	Возвращает поток, использующий вызывающую коллекцию в качестве источника для ввода-вывода элементов. В этом потоке поддерживаются последовательные операции ввода-вывода (добавлен в версии JDK 8)
<code>Object[] toArray()</code>	Возвращает массив, содержащий все элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции
<code><T> T[] toArray(T[] массив[])</code>	Возвращает массив, содержащий элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции. Если размер заданного <i>массива</i> равен количеству элементов в коллекции, они возвращаются в этом <i>массиве</i> . Если же размер <i>массива</i> меньше количества элементов в коллекции, то создается и возвращается новый массив нужного размера. А если размер <i>массива</i> больше количества элементов в коллекции, то во всех элементах, следующих за последним из коллекции, устанавливается пустое значение <code>null</code> . И если любой элемент коллекции относится к типу, не являющемуся подтипом <i>массива</i> , то генерируется исключение типа <code>ArrayStoreException</code>

Объекты вводятся в коллекции методом `add()`. Следует, однако, иметь в виду, что метод `add()` принимает аргумент типа *E*. Следовательно, добавляемые в коллекцию объекты должны быть совместимы с предполагаемым типом данных в коллекции. Вызвав метод `addAll()`, можно ввести все содержимое одной коллекции в другую.

Вызвав метод `remove()`, можно удалить из коллекции отдельный объект. Для того чтобы из коллекции удалить группу объектов, достаточно вызвать метод `removeAll()`. А для того чтобы удалить из коллекции все элементы, кроме указанных, следует вызвать метод `retainAll()`. В версии JDK 8 появилась возможность удалить элемент из коллекции, если он удовлетворяет условию, которое задается в качестве параметра предикат при вызове метода `removeIf()`. Этот параметр относится к типу функционального интерфейса `Predicate`, внедренного в версии JDK 8, как поясняется в главе 19. И наконец, для очистки коллекции достаточно вызвать метод `clear()`.

Имеется также возможность определить, содержит ли коллекция определенный объект, вызвав метод `contains()`. Чтобы определить, содержит ли одна коллекция все члены другой, следует вызвать метод `containsAll()`. А определить, пуста ли коллекция, можно с помощью метода `isEmpty()`. Количество элементов, содержащихся в данный момент в коллекции, возвращает метод `size()`.

Оба метода `toArray()` возвращают массив, который содержит элементы, хранящиеся в коллекции. Первый из них возвращает массив класса `Object`, а второй — массив элементов того же типа, что и массив, указанный в качестве параметра этого метода. Обычно второй метод более предпочтителен, поскольку он возвращает массив элементов нужного типа. Эти методы оказываются важнее, чем может показаться на первый взгляд. Ведь обрабатывать содержимое коллекции с использованием синтаксиса массивов нередко оказывается очень выгодно. Обеспечив связь коллекции с массивом, можно выгодно воспользоваться преимуществами обоих языковых средств Java.

Две коллекции можно сравнить на равенство, вызвав метод `equals()`. Точный смысл равенства может зависеть от конкретной коллекции. Например, метод `equals()` можно реализовать таким образом, чтобы он сравнивал значения элементов, хранимых в коллекции. В качестве альтернативы методу `equals()` можно сравнивать ссылки на эти элементы.

Еще один очень важный метод `iterator()` возвращает итератор коллекции. Новый метод `spliterator()` возвращает итератор-разделитель для коллекции. Итераторы очень часто используются для обращения с коллекциями. И наконец, методы `stream()` и `parallelStream()` возвращают поток в виде объекта интерфейса `Stream`, использующий коллекцию для ввода-вывода элементов (подробнее новый интерфейс `Stream` рассматривается в главе 29).

Интерфейс `List`

Этот интерфейс расширяет интерфейс `Collection` и определяет такое поведение коллекций, которое сохраняет последовательность элементов. Элементы могут быть введены или извлечены по индексу их позиции в списке, начиная с нуля. Список может содержать повторяющиеся элементы. Интерфейс `List` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые должен содержать список.

```
interface List<E>
```

Помимо методов, объявленных в интерфейсе `Collection`, в интерфейсе `List` определяется ряд своих методов, перечисленных в табл. 18.3. Однако некоторые из этих методов генерируют исключение типа `UnsupportedOperationException`, если коллекция не может быть видоизменена, а исключение типа `ClassCastException` генерируется, если объекты несовместимы, например, при попытке ввести в список элемент несовместимого типа. Кроме того, некоторые методы генерируют исключение типа `IndexOutOfBoundsException`, если указан неверный индекс. А исключение типа `NullPointerException` генерируется при попытке ввести в список пустой объект со значением `null`, когда пустые элементы в данном списке не допускаются. И наконец, исключение типа `IllegalArgumentException` генерируется при указании неверного аргумента.

Таблица 18.3. Методы из интерфейса List

Метод	Описание
<code>void add(int индекс, E объект)</code>	Вводит заданный <i>объект</i> на позиции вызывающего списка по указанному <i>индексу</i> . Любые введенные ранее элементы смещаются, начиная с указанной позиции и далее к началу списка. Это означает, что элементы в списке не перезаписываются
<code>boolean addAll(int индекс, Collection<? extends E> c)</code>	Вводит все элементы из коллекции <i>c</i> в вызывающий список, начиная с позиции по указанному <i>индексу</i> . Введенные ранее элементы смещаются, начиная с указанной позиции и далее к началу списка. Это означает, что элементы в списке не перезаписываются. Возвращает логическое значение <code>true</code> , если вызывающий список изменяется, а иначе – логическое значение <code>false</code>
<code>E get(int индекс)</code>	Возвращает объект, хранящийся в вызывающем списке на позиции по указанному <i>индексу</i>
<code>int indexOf(Object объект)</code>	Возвращает индекс первого экземпляра заданного <i>объекта</i> в вызывающем списке. Если заданный <i>объект</i> отсутствует в списке, возвращается значение <code>-1</code>
<code>int lastIndexOf(Object объект)</code>	Возвращает индекс последнего экземпляра заданного <i>объекта</i> в вызывающем списке. Если заданный <i>объект</i> отсутствует в списке, возвращается значение <code>-1</code>
<code>ListIterator<E> listIterator()</code>	Возвращает итератор для обхода элементов с начала вызывающего списка
<code>ListIterator<E> listIterator(int индекс)</code>	Возвращает итератор для обхода элементов вызывающего списка, начиная с позиции по указанному <i>индексу</i>
<code>E remove(int индекс)</code>	Удаляет элемент из вызывающего списка на позиции по указанному <i>индексу</i> и возвращает удаленный элемент. Результатирующий список уплотняется, т.е. элементы, следующие за удаленным, смещаются на одну позицию назад
<code>default void replaceAll(UnaryOperator<E> opToApply)</code>	Обновляет каждый элемент списка значением, получаемым из функции, определяемой параметром <i>opToApply</i> (добавлен в версии JDK 8)
<code>E set(int индекс, E объект)</code>	Присваивает заданный <i>объект</i> элементу, находящемуся в списке на позиции по указанному <i>индексу</i> . Возвращает прежнее значение
<code>default void sort(Comparator<super E> компаратор)</code>	Сортирует список, используя заданный <i>компаратор</i> (добавлен в версии JDK 8)
<code>List<E> subList(int начало, int конец)</code>	Возвращает список, включающий элементы от позиции <i>начало</i> до позиции <i>конец-1</i> из вызывающего списка. Ссылки на элементы из возвращаемого списка сохраняются и в вызывающем списке

Варианты методов `add()` и `addAll()`, определенные в интерфейсе `Collection`, дополняются в интерфейсе `List` методами `add(int, E)` и `addAll(int, Collection)`. Эти методы вводят элементы на позиции по указанному индексу. Кроме того, семантика методов `add(E)` и `addAll(Collection)`, определенная в интерфейсе `Collection`, изменяется в интерфейсе `List` таким образом, что они вводят элементы в конце списка. Изменить каждый элемент в коллекции можно с помощью метода `replaceAll()`. (Для этой цели в версии JDK 8 внедрен функциональный интерфейс `UnaryOperator`, как поясняется в главе 19.)

Из данного списка можно получить подсписок, вызвав метод `subList()` и указав начальный и конечный индексы подсписка. Нетрудно догадаться, что благодаря методу `subList()` обращаться со списками намного удобнее. Отсортировать список можно, в частности, с помощью метода `sort()`, определенного в интерфейсе `List`.

Интерфейс `Set`

В интерфейсе `Set` определяется множество. Он расширяет интерфейс `Collection` и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод `add()` возвращает логическое значение `false` при попытке ввести в множество дублирующий элемент. В этом интерфейсе не определяется никаких дополнительных методов. Интерфейс `Set` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые должно содержать множество.

```
interface Set<E>
```

Интерфейс `SortedSet`

Интерфейс `SortedSet` расширяет интерфейс `Set` и определяет поведение множеств, отсортированных в порядке возрастания. Интерфейс `SortedSet` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые должно содержать множество.

```
interface SortedSet<E>
```

Помимо методов, предоставляемых интерфейсом `Set`, в интерфейсе `SortedSet` объявляются методы, перечисленные в табл. 18.4. Некоторые из них генерируют исключение типа `NoSuchElementException`, если в вызывающем множестве отсутствуют какие-нибудь элементы. Исключение типа `ClassCastException` генерируется, если заданный объект несовместим с элементами множества. Исключение типа `NullPointerException` генерируется при попытке использовать пустой объект, когда пустое значение `null` в множестве недопустимо. А при указании неверного аргумента генерируется исключение типа `IllegalArgumentException`.

В интерфейсе `SortedSet` определен ряд методов, упрощающих обработку элементов множеств. Чтобы получить первый элемент в отсортированном множестве, достаточно вызвать метод `first()`, а чтобы получить последний элемент — метод `last()`. Из отсортированного множества можно получить подмножество, вызвав метод `subSet()` и указав первый и последний элементы множества. Если

требуется получить подмножество, которое начинается с первого элемента существующего множества, следует вызывать метод `headSet()`. А если требуется получить подмножество, которое начинается с последнего элемента существующего множества, следует вызывать метод `tailSet()`.

Таблица 18.4. Методы из интерфейса `SortedSet`

Метод	Описание
<code>Comparator<? super E></code> <code>comparator()</code>	Возвращает компаратор отсортированного множества. Если для множества выбирается естественный порядок сортировки, то возвращается пустое значение <code>null</code>
<code>E first()</code>	Возвращает первый элемент вызывающего отсортированного множества
<code>SortedSet<E></code> <code>headSet(E конец)</code>	Возвращает объект типа <code>SortedSet</code> , содержащий элементы из вызывающего отсортированного множества, которые предшествуют элементу, определяемому параметром <code>конец</code> . Ссылки на элементы из возвращаемого отсортированного множества сохраняются и в вызывающем отсортированном множестве
<code>E last()</code>	Возвращается последний элемент из вызывающего отсортированного множества
<code>SortedSet<E></code> <code>subSet(E начало, E конец)</code>	Возвращает объект типа <code>SortedSet</code> , который включает в себя элементы, начиная с позиции <code>начало</code> и до позиции <code>конец-1</code> . Ссылки на элементы из возвращаемого отсортированного множества сохраняются и в вызывающем отсортированном множестве
<code>SortedSet<E></code> <code>tailSet(E начало)</code>	Возвращает объект типа <code>SortedSet</code> , содержащий элементы из вызывающего множества, которые следуют после элемента на заданной позиции <code>начало</code> . Ссылки на элементы из возвращаемого отсортированного множества сохраняются и в вызывающем отсортированном множестве

Интерфейс `NavigableSet`

Этот интерфейс расширяет интерфейс `SortedSet` и определяет поведение коллекции, извлечение элементов из которой осуществляется на основании наиболее точного совпадения с заданным значением или несколькими значениями. Интерфейс `NavigableSet` является обобщенным и объявляется следующим образом:

```
interface NavigableSet<E>
```

где `E` обозначает тип объектов, содержащихся в множестве. Помимо методов, наследуемых из интерфейса `SortedSet`, в интерфейсе `NavigableSet` определяются методы, перечисленные в табл. 18.5. Они генерируют исключение типа `ClassCastException`, если заданный объект несовместим с элементами множества. Исключение типа `NullPointerException` генерируется при попытке ввести пустой объект, когда в множестве не допускаются пустые значения `null`. А при указании неверного аргумента передается исключение типа `IllegalArgumentException`.

Таблица 18.5. Методы из интерфейса NavigableSet

Метод	Описание
<code>E ceiling (E объект)</code>	Выполняет поиск в множестве наименьшего элемента <i>e</i> по критерию <i>e</i> \geq <i>объект</i> . Если такой элемент найден, он возвращается, в противном случае – пустое значение <code>null</code>
<code>Iterator<E> descendingIterator ()</code>	Возвращает итератор, выполняющий обход от большего элемента множества к меньшему, т.е. обратный итератор
<code>NavigableSet<E> descendingSet ()</code>	Возвращает объект типа <code>NavigableSet</code> , обратный по отношению к вызывающему множеству. Результатирующее множество поддерживается вызывающим множеством
<code>E floor (E объект)</code>	Выполняет поиск в множестве наибольшего элемента <i>e</i> по критерию <i>e</i> \leq <i>объект</i> . Если такой элемент найден, он возвращается, в противном случае – пустое значение <code>null</code>
<code>NavigableSet<E> headSet (E верхняя_граница, boolean включительно)</code>	Возвращает объект типа <code>NavigableSet</code> , содержащий все элементы вызывающего множества, меньше заданной <i>верхней_границы</i> . Если же параметр <i>включительно</i> принимает логическое значение <code>true</code> , то в возвращаемое множество включается и элемент, равный заданной <i>верхней_границе</i> . Результатирующее множество поддерживается вызывающим множеством
<code>E higher (E объект)</code>	Выполняет поиск в множестве наибольшего элемента <i>e</i> по критерию <i>e</i> $>$ <i>объект</i> . Если такой элемент найден, он возвращается, а иначе – пустое значение <code>null</code>
<code>E lower (E объект)</code>	Выполняет поиск в множестве наименьшего элемента <i>e</i> по критерию <i>e</i> $<$ <i>объект</i> . Если такой элемент найден, он возвращается, а иначе – пустое значение <code>null</code>
<code>E pollFirst ()</code>	Возвращает первый элемент, попутно удаляя его из множества. Это элемент с наименьшим значением, поскольку множество отсортировано. Если же множество оказывается пустым, то возвращается пустое значение <code>null</code>
<code>E pollLast ()</code>	Возвращает последний элемент, попутно удаляя его из множества. Это элемент с наибольшим значением, поскольку множество отсортировано. Если же множество оказывается пустым, то возвращается пустое значение <code>null</code>
<code>NavigableSet<E> subSet (E нижняя_граница, boolean включая_нижнюю_границу, E верхняя_граница, boolean включая_верхнюю_границу)</code>	Возвращает объект типа <code>NavigableSet</code> , включающий все элементы вызывающего множества, которые больше заданной <i>нижней_границы</i> и меньше заданной <i>верхней_границы</i> . Если же параметр <i>включая_нижнюю_границу</i> принимает логическое значение <code>true</code> , то в результатирующем множестве включается элемент, равный заданной <i>нижней_границе</i> . А если параметр <i>включая_верхнюю_границу</i> принимает логическое значение <code>true</code> , то в результатирующем множестве включается и элемент, равный заданной <i>верхней_границе</i> . Результатирующее множество поддерживается вызывающим множеством

Окончание табл. 18.5

Метод	Описание
<code>NavigableSet<E> tailSet(E нижняя_граница, boolean включительно)</code>	Возвращает объект типа <code>NavigableSet</code> , включающий все элементы из вызывающего множества, которые больше заданной <i>нижней_границы</i> . Если же параметр <i>включительно</i> принимает логическое значение <code>true</code> , то в результирующее множество включается элемент, равный заданной <i>нижней_границе</i> . Результирующее множество поддерживается вызывающим множеством

Интерфейс Queue

Этот интерфейс расширяет интерфейс `Collection` и определяет поведение очереди, которая действует как список по принципу “первым вошел – первым обслужен”. Тем не менее имеются разные виды очередей, порядок организации в которых основывается на некотором критерии. Интерфейс `Queue` является обобщенным и объявляется следующим образом:

```
interface Queue<E>
```

где `E` обозначает тип объектов, которые будут храниться в очереди. Методы, определенные в интерфейсе `Queue`, перечислены в табл. 18.6.

Таблица 18.6. Методы из интерфейса `Queue`

Метод	Описание
<code>E element()</code>	Возвращает элемент из головы очереди. Возвращаемый элемент не удаляется. Если очередь пуста, генерируется исключение типа <code>NoSuchElementException</code>
<code>boolean offer(E объект)</code>	Пытается ввести заданный <i>объект</i> в очередь. Возвращает логическое значение <code>true</code> , если <i>объект</i> введен, а иначе – логическое значение <code>false</code>
<code>E peek()</code>	Возвращает элемент из головы очереди. Если очередь пуста, возвращает пустое значение <code>null</code> . Возвращаемый элемент не удаляется из очереди
<code>E poll()</code>	Возвращает элемент из головы очереди и удаляет его. Если очередь пуста, возвращает пустое значение <code>null</code> .
<code>E remove()</code>	Удаляет элемент из головы очереди, возвращая его. Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста

Некоторые методы из данного интерфейса генерируют исключение типа `ClassCastException`, если заданный объект несовместим с элементами очереди. Исключение типа `NullPointerException` генерируется, когда предпринимается попытка сохранить пустой объект, а пустые элементы в очереди не разрешены. Исключение типа `IllegalArgumentException` генерируется при указании неверного аргумента. Исключение типа `IllegalStateException` генерируется при попытке ввести объект в заполненную очередь фиксированной длины. И наконец,

исключение типа `NoSuchElementException` генерируется при попытке удалить элемент из пустой очереди.

Несмотря на всю свою простоту, интерфейс `Queue` представляет интерес с нескольких точек зрения. Во-первых, элементы могут удаляться только из начала очереди. Во-вторых, имеются два метода, `poll()` и `remove()`, с помощью которых можно получать и удалять элементы из очереди. Отличаются они тем, что метод `poll()` возвращает пустое значение `null`, если очередь пуста, тогда как метод `remove()` генерирует исключение. И в-третьих, имеются еще два метода, `element()` и `peek()`, которые получают элемент из головы очереди, но не удаляют его. Отличаются они тем, что при пустой очереди метод `element()` генерирует исключение, тогда как метод `peek()` возвращает пустое значение `null`. И наконец, следует иметь в виду, что метод `offer()` только пытается ввести элемент в очередь. А поскольку некоторые очереди имеют фиксированную длину и могут быть заполнены, то вызов метода `offer()` может завершиться неудачно.

Интерфейс `Dequeue`

Интерфейс `Dequeue` расширяет интерфейс `Queue` и определяет поведение двусторонней очереди, которая может функционировать как стандартная очередь по принципу “первым вошел – первым обслужен” или как стек по принципу “последним вошел – первым обслужен”. Интерфейс `Dequeue` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые будет содержать двусторонняя очередь.

`interface Dequeue<E>`

Помимо методов, наследуемых из интерфейса `Queue`, в интерфейсе `Dequeue` определяются методы, перечисленные в табл. 18.7. Некоторые из этих методов генерируют исключение типа `ClassCastException`, если заданный объект несовместим с элементами двусторонней очереди. Исключение типа `NullPointerException` генерируется, когда предпринимается попытка сохранить пустой объект, а пустые элементы двусторонней очереди не допускаются. При указании неверного аргумента генерируется исключение типа `IllegalArgumentException`. Исключение типа `IllegalStateException` генерируется при попытке ввести объект в заполненную двустороннюю очередь фиксированной длины. И наконец, исключение типа `NoSuchElementException` генерируется при попытке удалить элемент из пустой очереди.

Таблица 18.7. Методы из интерфейса `Dequeue`

Метод	Описание
<code>void addFirst (E объект)</code>	Вводит заданный <i>объект</i> в голову двусторонней очереди. Генерирует исключение типа <code>IllegalStateException</code> , если в очереди фиксированной длины нет свободного места
<code>void addLast (E объект)</code>	Вводит заданный <i>объект</i> в хвост двусторонней очереди. Генерирует исключение типа <code>IllegalStateException</code> , если в очереди фиксированной длины нет свободного места

Продолжение табл. 18.7

Метод	Описание
<code>Iterator<E> descendingIterator()</code>	Возвращает итератор для обхода элементов от хвоста к голове двусторонней очереди. Иными словами, возвращает обратный итератор
<code>E getFirst()</code>	Возвращает первый элемент двусторонней очереди. Возвращаемый элемент из очереди не удаляется. Генерирует исключение типа <code>NoSuchElementException</code> , если двусторонняя очередь пуста
<code>E getLast()</code>	Возвращает последний элемент двусторонней очереди. Возвращаемый элемент из очереди не удаляется. Генерирует исключение типа <code>NoSuchElementException</code> , если двусторонняя очередь пуста
<code>boolean offerFirst(E объект)</code>	Пытается ввести заданный <i>объект</i> в голову двусторонней очереди. Возвращает логическое значение <code>true</code> , если <i>объект</i> введен, а иначе – логическое значение <code>false</code> . Таким образом, этот метод возвращает логическое значение <code>false</code> при попытке ввести заданный <i>объект</i> в заполненную двустороннюю очередь фиксированной длины
<code>boolean offerLast(E объект)</code>	Пытается ввести заданный <i>объект</i> в хвост двусторонней очереди. Возвращает логическое значение <code>true</code> , если <i>объект</i> введен, а иначе – логическое значение <code>false</code>
<code>E peekFirst()</code>	Возвращает элемент, находящийся в голове двусторонней очереди. Если очередь пуста, возвращает пустое значение <code>null</code> . Возвращаемый элемент из очереди не удаляется
<code>E peekLast()</code>	Возвращает элемент, находящийся в хвосте двусторонней очереди. Если очередь пуста, возвращает пустое значение <code>null</code> . Возвращаемый элемент из очереди не удаляется
<code>E pollFirst()</code>	Возвращает элемент, находящийся в голове двусторонней очереди, одновременно удаляя его из очереди. Если очередь пуста, возвращает пустое значение <code>null</code>
<code>E pollLast()</code>	Возвращает элемент, находящийся в хвосте двусторонней очереди, одновременно удаляя его из очереди. Если очередь пуста, возвращает пустое значение <code>null</code>
<code>E pop()</code>	Возвращает элемент, находящийся в голове двусторонней очереди, одновременно удаляя его из очереди. Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста
<code>void push(E объект)</code>	Вводит заданный <i>объект</i> в голову двусторонней очереди. Если в очереди фиксированной длины нет свободного места, генерирует исключение типа <code>IllegalStateException</code>
<code>E removeFirst()</code>	Возвращает элемент из головы двусторонней очереди, одновременно удаляя его из очереди. Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста

Окончание табл. 18.7

Метод	Описание
<code>boolean removeFirstOccurrence (Object объект)</code>	Удаляет первый экземпляр заданного <i>объекта</i> из очереди. Возвращает логическое значение <code>true</code> при удачном исходе операции, а если двусторонняя очередь не содержит заданный <i>объект</i> – логическое значение <code>false</code>
<code>E removeLast()</code>	Возвращает элемент из хвоста двусторонней очереди, одновременно удаляя его из очереди. Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста
<code>boolean removeLastOccurrence (Object объект)</code>	Удаляет последний экземпляр заданного <i>объекта</i> из очереди. Возвращает логическое значение <code>true</code> при удачном исходе операции, а если двусторонняя очередь не содержит заданный <i>объект</i> – логическое значение <code>false</code>

Обратите внимание на то, что в состав интерфейса `Deque` входят методы `push()` и `pop()`, благодаря которым этот интерфейс может функционировать как стек. Обратите также внимание на метод `descendingIterator()`, возвращающий итератор, который обходит элементы очереди в обратном порядке, т.е. от хвоста очереди к ее голове (или конца данного вида коллекции к ее началу). Реализация интерфейса `Deque` в виде двусторонней очереди может быть *ограниченной по емкости*, т.е. в такую очередь может быть введено ограниченное количество элементов. В этом случае попытка ввести элемент в очередь может оказаться неудачной. Неудачный исход подобных операций интерпретируется в интерфейсе `Deque` двумя способами. Во-первых, методы вроде `addFirst()` и `addLast()` генерируют исключение типа `IllegalStateException`, если двусторонняя очередь имеет ограниченную емкость. И во-вторых, методы наподобие `offerFirst()` и `offerLast()` возвращают логическое значение `false`, если элемент не может быть введен в очередь.

Классы коллекций

А теперь, когда представлены интерфейсы коллекций, можно приступить к рассмотрению стандартных классов, которые их реализуют. Одни из этих классов предоставляют полную реализацию соответствующих интерфейсов и могут применяться без изменений. А другие являются абстрактными, предоставляя только шаблонные реализации соответствующих интерфейсов, которые используются в качестве отправной точки для создания конкретных коллекций. Как правило, классы коллекций не синхронизированы, но если требуется, то можно получить их синхронизированные варианты, как показано далее в этой главе. Базовые классы коллекций перечислены в табл. 18.8.

В последующих разделах рассматриваются конкретные классы коллекций и демонстрируется их применение.

На заметку! Помимо классов коллекций, некоторые классы, унаследованные из прежних версий, например `Vector`, `Stack` и `HashTable`, были переделаны для поддержки коллекций. Они также рассматриваются далее в этой главе.

Таблица 18.8. Базовые классы коллекций

Класс	Описание
AbstractCollection	Реализует большую часть интерфейса Collection
AbstractList	Расширяет класс AbstractCollection и реализует большую часть интерфейса List
AbstractQueue	Расширяет класс AbstractCollection и реализует отдельные части интерфейса Queue
AbstractSequentialList	Расширяет класс AbstractList для применения в коллекциях, использующих последовательности вместо случайного доступа к элементам
LinkedList	Реализует связный список, расширяя класс AbstractSequentialList
ArrayList	Реализует динамический массив, расширяя класс AbstractList
ArrayDeque	Реализует динамическую двухстороннюю очередь, расширяя класс AbstractCollection и реализуя интерфейс Deque
AbstractSet	Расширяет класс AbstractCollection и реализует большую часть интерфейса Set
EnumSet	Расширяет класс AbstractSet для применения вместе с элементами типа enum
HashSet	Расширяет класс AbstractSet для применения вместе с хеш-таблицами
LinkedHashSet	Расширяет класс HashSet , разрешая итерацию с вводом элементов в определенном порядке
PriorityQueue	Расширяет класс AbstractQueue для поддержки очередей по приоритетам
TreeSet	Реализует множество, хранимое в древовидной структуре. Расширяет класс AbstractSet

Класс **ArrayList**

Класс **ArrayList** расширяет класс **AbstractList** и реализует интерфейс **List**. Класс **ArrayList** является обобщенным и объявляется приведенным ниже образом, где параметр **E** обозначает тип сохраняемых объектов.

```
class ArrayList<E>
```

В классе **ArrayList** поддерживаются динамические массивы, которые могут наращиваться по мере надобности. Стандартные массивы в Java имеют фиксированную длину. После того как массив создан, он не может увеличиваться или уменьшаться, а следовательно, нужно заранее знать, сколько элементов требуется в нем хранить. Но иногда еще до стадии выполнения неизвестно, насколько большой

массив потребуется. В качестве выхода из данного положения в каркасе коллекций определяется класс `ArrayList`. По существу, класс `ArrayList` представляет собой списочный массив объектных ссылок переменной длины. Это означает, что размер объекта типа `ArrayList` может динамически увеличиваться или уменьшаться. Списочные массивы создаются с некоторым начальным размером. Когда же этого первоначального размера оказывается недостаточно, коллекция автоматически расширяется. А когда из коллекции удаляются объекты, она может сокращаться.

В классе `ArrayList` определены следующие конструкторы:

```
ArrayList()
ArrayList(Collection <? extends E> c)
ArrayList(int емкость)
```

Первый конструктор создает пустой списочный массив, второй — списочный массив, инициализируемый элементами из заданной коллекции *c*, а третий — списочный массив, имеющий начальную емкость. Под *емкостью* здесь подразумевается размер базового массива, используемого для хранения элементов данного вида коллекции. Емкость наращивается автоматически по мере ввода элементов в списочный массив.

В следующем примере программы демонстрируется простое применение класса `ArrayList`. В этой программе сначала создается списочный массив объектов типа `String`, затем в него вводится несколько символьных строк. (Напомним, что символьные строки, заключенные в кавычки, преобразуются в объекты типа `String`.) Полученный в итоге список символьных строк выводится на экран. Некоторые элементы удаляются из этого списка, после чего он выводится снова.

```
/// Продемонстрировать применение класса ArrayList
import java.util.*;

class ArrayListDemo {
    public static void main (String args[]) {
        // создать списочный массив
        ArrayList<String> al = new ArrayList<String>();

        System.out.println("Начальный размер списочного массива al: " + al.size());

        // ввести элементы в списочный массив
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1, "A2");
        System.out.println("Размер списочного массива al после ввода элементов: " +
                           al.size());

        // вывести списочный массив
        System.out.println("Содержимое списочного массива al: " + al);

        // удалить элементы из списочного массива
        al.remove("F");
        al.remove(2);
```

```

        System.out.println(
            "Размер списочного массива al после удаления элементов: " +
            al.size());
    }
}

```

Ниже приведен результат, выводимый данной программой. Обратите внимание на то, что списочный массив `al` изначально пуст и увеличивается по мере ввода в него элементов. Когда же элементы удаляются из списочного массива, его размер сокращается.

```

Начальный размер списочного массива al: 0
Размер списочного массива al после ввода элементов: 7
Содержимое списочного массива al: [C, A2, A, E, B, D, F]
Размер списочного массива al после удаления элементов: 5
Содержимое списочного массива al: [C, A2, E, B, D]

```

В приведенном выше примере содержимое коллекции выводится с преобразованием типов, выполняемым по умолчанию методом `toString()`, который наследуется от класса `AbstractCollection`. И хотя этот способ удобен для написания коротких примеров программ, на практике им редко пользуются для вывода содержимого настоящих коллекций. Обычно для этой цели программисты представляют свои процедуры вывода. Но для нескольких последующих примеров вполне подходит вывод, выполняемый методом `toString()` по умолчанию.

Несмотря на то что емкость объектов типа `ArrayList` наращивается автоматически, ее можно увеличивать и вручную, вызывая метод `ensureCapacity()`. Это может потребоваться в том случае, если заранее известно, что в коллекции предполагается сохранить намного больше элементов, чем она содержит в данный момент. Увеличив емкость списочного массива в самом начале его обработки, можно избежать дополнительного перераспределения оперативной памяти впоследствии. Ведь перераспределение оперативной памяти – дорогостоящая операция с точки зрения затрат времени, и поэтому исключение лишних операций подобного рода способствует повышению производительности. Ниже приведена общая форма метода `ensureCapacity()`, где параметр `емкость` обозначает новую минимальную емкость коллекции.

```
void ensureCapacity(int емкость)
```

С другой стороны, если требуется уменьшить размер базового массива, на основе которого строится объект типа `ArrayList`, до текущего количества хранящихся в действительности объектов, следует вызвать метод `trimToSize()`. Ниже приведена общая форма этого метода.

```
void trimToSize()
```

Получение массива из коллекции типа `ArrayList`

При обработке списочного массива типа `ArrayList` иногда требуется получить обычный массив, содержащий все элементы списка. Это можно сделать, вызвав метод `toArray()`, определенный в интерфейсе `Collection`.

Имеется несколько причин, по которым возникает потребность преобразовать коллекцию в массив.

- Ускорение выполнения некоторых операций.
- Передача массива в качестве параметра методам, которые не перегружаются, чтобы принимать коллекции непосредственно.
- Интеграция нового кода, основанного на коллекциях, с унаследованным кодом, который не распознает коллекции.

Независимо от конкретной причины преобразовать коллекцию типа `ArrayList` в массив не составляет особого труда. Как пояснялось ранее, имеются два варианта метода `toArray()`, общие формы которых приведены ниже.

```
Object[] toArray()
<T> T[] toArray(T[] массив[])
```

В первой форме метод `toArray()` возвращает массив объектов типа `Object`, а во второй форме – массив элементов, относящихся к типу `T`. Обычно вторая форма данного метода удобнее, поскольку в ней возвращается надлежащий тип массива. В следующем примере программы демонстрируется применение именно этой формы метода `toArray()`.

```
// Преобразовать списочный массив ArrayList в обычный массив
import java.util.*;

class ArrayListToArray {
    public static void main(String args[]) {
        // создать списочный массив
        ArrayList<Integer> al = new ArrayList<Integer>();

        // ввести элементы в списочный массив
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);

        System.out.println("Содержимое списочного массива al: " + al);

        // получить обычный массив
        Integer ia[] = new Integer[al.size()];
        ia = al.toArray(ia);

        int sum = 0;
        // суммировать элементы массива
        for(int i : ia) sum += i;

        System.out.println("Сумма: " + sum);
    }
}
```

Данная программа выводит следующий результат:

```
Содержимое списочного массива al: [1, 2, 3, 4]
Сумма: 10
```

Эта программа начинается с создания коллекции целых чисел. Затем вызывается метод `toArray()` и получается массив элементов типа `Integer`. Далее содержимое массива суммируется в цикле `for` в стиле `for each`. У этой программы имеется еще одна любопытная особенность. Как вам должно быть уже известно, коллекции могут содержать только ссылки, а не значения примитивных типов. Но автоматическая упаковка позволяет передавать методу `add()` значения типа `int`, не прибегая к необходимости заключать их в оболочку типа `Integer`, как это демонстрируется в данной программе. Таким образом, автоматическая упаковка ощутимо облегчает сохранение в коллекциях значений примитивных типов.

Класс `LinkedList`

Этот класс расширяет класс `AbstractSequentialList` и реализует интерфейсы `List`, `Deque` и `Queue`. Он предоставляет структуру данных связного списка. Класс `LinkedList` является обобщенным и объявляется следующим образом:

```
class LinkedList<E>
```

где `E` обозначает тип сохраняемых в списке объектов. У класса `LinkedList` имеются два конструктора:

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

Первый конструктор создает пустой связный список, а второй – связный список и инициализирует его содержимым коллекции `c`. В классе `LinkedList` реализуется интерфейс `Deque`, и благодаря этому становятся доступными методы, определенные в интерфейсе `Deque`. Например, чтобы ввести элементы в начале списка, достаточно вызвать метод `addFirst()` или `offerFirst()`, а для того чтобы ввести элементы в конце списка – метод `addLast()` или `offerLast()`. Чтобы получить первый элемент из списка, следует вызвать метод `getFirst()` или `peekFirst()`, а для того чтобы удалить первый элемент из списка – метод `removeFirst()` или `pollFirst()`. И наконец, чтобы получить последний элемент из списка, следует вызвать метод `getLast()` или `peekLast()`, а для того чтобы удалить последний элемент из списка – метод `removeLast()` или `pollLast()`.

В следующем примере программы демонстрируется применение класса `LinkedList`:

```
// Продемонстрировать применение класса LinkedList
import java.util.*;

class LinkedListDemo {
    public static void main(String args[]) {
        // создать связный список
        LinkedList<String> ll = new LinkedList<String>();

        // ввести элементы в связный список
        ll.add("F");
        ll.add("B");
        ll.add("D");
        ll.add("E");
        ll.add("C");
        ll.addLast("Z");
```

```

ll.addFirst("A");
ll.add(1, "A2");
System.out.println(
    "Исходное содержимое связного списка ll: " + ll);
// удалить элементы из связного списка
ll.remove("F");
ll.remove(2);
System.out.println(
    "Содержимое связного списка ll " +
    "после удаления элементов: " + ll);
// удалить первый и последний элементы из связного списка
ll.removeFirst();
ll.removeLast();

System.out.println(
    "Содержимое связного списка ll после удаления " +
    "первого и последнего элементов: "+ ll);

// получить и присвоить значение
String val = ll.get(2);
ll.set(2, val + " изменено");

System.out.println(
    "Содержимое связного списка ll после изменения: " + ll);
}
}

```

Ниже приведен результат, выводимый данной программой.

```

Исходное содержимое связного списка ll: [A, A2, F, B, D, E, C, Z]
Содержимое связного списка ll после удаления элементов: [A, A2, D, E, C, Z]
Содержимое связного списка ll после удаления первого и последнего
элементов: [A2, D, E, C]
Содержимое связного списка ll после изменения: [A2, D, E изменено, C]

```

В классе `LinkedList` реализуется интерфейс `List`, и поэтому в результате вызова метода `add(E)` элементы вводятся в конце списка, как это делается и при вызове метода `addLast()`. Чтобы ввести элементы в определенном месте списка, следует воспользоваться формой метода `add(int, E)`, как продемонстрировано выше на примере вызова `add(1, "A2")`.

Обратите внимание, как третий элемент связного списка `ll` изменяется с помощью методов `get()` и `set()`. Чтобы получить текущее значение элемента, методу `get()` передается индекс позиции, на которой расположен нужный элемент. А для того чтобы присвоить новое значение элементу на этой позиции, методу `set()` передается соответствующий индекс и новое значение.

Класс `HashSet`

Класс `HashSet` расширяет класс `AbstractSet` и реализует интерфейс `Set`. Он служит для создания коллекции, для хранения элементов которой используется хеш-таблица. Класс `HashSet` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые будут храниться в хеш-множестве.

```
class HashSet<E>
```

Как известно, для хранения данных в хеш-таблице применяется механизм так называемого *хеширования*, где содержимое ключа служит для определения однозначного значения, называемого *хеш-кодом*. Этот хеш-код служит далее в качестве индекса, по которому сохраняются данные, связанные с ключом. Преобразование ключа в хеш-код выполняется автоматически, хотя сам хеш-код недоступен. Кроме того, в прикладном коде нельзя индексировать хеш-таблицу непосредственно. Преимущество хеширования заключается в том, что оно обеспечивает постоянство времени выполнения методов `add()`, `contains()`, `remove()` и `size()` – даже для крупных множеств.

В классе `HashSet` определены следующие конструкторы:

```
HashSet()
HashSet(Collection<? extends E> c)
HashSet(int емкость)
HashSet(int емкость, float коэффициент_заполнения)
```

В первой форме конструктора хеш-множество создается по умолчанию. Во второй форме хеш-множество инициируется содержимым заданной коллекции *c*. В третьей форме задается емкость хеш-множества (по умолчанию – 16), а в четвертой в качестве аргументов конструктора задается емкость хеш-множества и *коэффициент_заполнения*, иначе называемый *емкостью загрузки*. Коэффициент заполнения должен быть в пределах от 0,0 до 1,0, которые определяют, насколько заполненным должно быть хеш-множество, прежде чем будет изменен его размер. В частности, когда количество элементов становится больше емкости хеш-множества, умноженной на коэффициент заполнения, такое хеш-множество расширяется. В конструкторах, которые не принимают коэффициент заполнения в качестве параметра, выбирается значение этого коэффициента, равное 0,75.

В классе `HashSet` не определяются никаких дополнительных методов, помимо тех, что предоставляют его суперклассы и интерфейсы. Следует также иметь в виду, что класс `HashSet` не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств. Если же требуются сортированные множества, то для этой цели лучше выбрать другой вид коллекции, например `TreeSet`. Ниже приведен пример программы, демонстрирующей применение класса `HashSet`.

```
// Продемонстрировать применение класса HashSet
import java.util.*;

class HashSetDemo {
    public static void main(String args[]) {
        // создать хеш-множество
        HashSet<String> hs = new HashSet<String>();

        // ввести элементы в хеш-множество
        hs.add("Бета");
        hs.add("Альфа");
        hs.add("Эта");
        hs.add("Гамма");
        hs.add("Эпсилон");
        hs.add("Омега");

        System.out.println(hs);
    }
}
```

Ниже приведен результат, выводимый данной программой. Как пояснялось ранее, элементы не сохраняются в хеш-множестве в отсортированном порядке, поэтому порядок их вывода может варьироваться.

[Гамма, Эта, Альфа, Эпсилон, Омега, Бета]

Класс `LinkedHashSet`

Класс `LinkedHashSet` расширяет класс `HashSet`, не добавляя никаких новых методов. Этот класс является обобщенным и объявляется следующим образом:

```
class LinkedHashSet<E>
```

где `E` обозначает тип объектов, которые будут храниться в хеш-множестве. У этого класса такие же конструкторы, как и у класса `HashSet`.

В классе `LinkedHashSet` поддерживается связный список элементов хеш-множества в том порядке, в каком они введены в него. Это позволяет организовать итерацию с вводом элементов в определенном порядке. Следовательно, когда перебор элементов хеш-множества типа `LinkedHashSet` производится с помощью итератора, элементы извлекаются из этого множества в том порядке, в каком они были введены. Именно в этом порядке они будут также возвращены методом `toString()`, вызываемым для объекта типа `LinkedHashSet`. Чтобы увидеть эффект от применения класса `LinkedHashSet`, попробуйте подставить его в исходный код предыдущего примера программы вместо класса `HashSet`. После этого выводимый программой результат будет выглядеть так, как показано ниже, отражая тот порядок, в каком элементы были введены в хеш-множество.

[Бета, Альфа, Эта, Гамма, Эпсилон, Омега]

Класс `TreeSet`

Класс `TreeSet` расширяет класс `AbstractSet` и реализует интерфейс `NavigableSet`. Он создает коллекцию, где для хранения элементов применяется древовидная структура. Объекты сохраняются в отсортированном порядке по нарастающей. Время доступа и извлечения элементов достаточно мало, благодаря чему класс `TreeSet` оказывается отличным выбором для хранения больших объемов отсортированных данных, которые должны быть быстро найдены. Класс `TreeSet` является обобщенным классом и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые будут храниться в древовидном множестве.

```
class TreeSet<E>
```

В классе `TreeSet` определены следующие конструкторы:

```
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> компаратор)
TreeSet(SortedSet<E> ss)
```

В первой форме конструктора создается пустое древовидное множество, элементы которого будут отсортированы в естественном порядке по нарастающей. Во второй форме создается древовидное множество, содержащее элементы заданной кол-

лекции с. В третьей форме создается пустое древовидное множество, элементы которого будут отсортированы заданным компаратором. (Компараторы рассматриваются далее в этой главе.) И наконец, в четвертой форме создается древовидное множество, содержащее элементы заданного отсортированного множества *ss*. В приведенном ниже примере программы демонстрируется применение класса *TreeSet*.

```
// Продемонстрировать применение класса TreeSet
import java.util.*;

class TreeSetDemo {
    public static void main(String args[]) {
        // создать древовидное множество типа TreeSet
        TreeSet<String> ts = new TreeSet<String>();

        // ввести элементы в древовидное множество типа TreeSet
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");

        System.out.println(ts);
    }
}
```

Эта программа выводит следующий результат:

[A, B, C, D, E, F]

Как пояснялось ранее, класс *TreeSet* сохраняет элементы в древовидной структуре. Они автоматически располагаются в отсортированном порядке, что и подтверждает выводимый программой результат. А поскольку класс *TreeSet* реализует интерфейс *NavigableSet*, внедренный в версии Java SE 6, то для извлечения элементов из древовидного множества типа *TreeSet* становятся доступными методы, определенные в интерфейсе *NavigableSet*. Допустим, в исходный код программы из предыдущего примера была добавлена следующая строка кода, где для получения множества *ts*, содержащего элементы от C (включительно) до F (исключительно), сначала вызывается метод *subSet()*, а затем выводится результирующее множество:

```
System.out.println(ts.subSet("C", "F"));
```

Ниже приведен результат выполнения этой строки кода. При желании можете поэкспериментировать с другими методами, определенными в интерфейсе *NavigableSet*.

[C, D, E]

Класс *PriorityQueue*

Класс *PriorityQueue* расширяет класс *AbstractQueue* и реализует интерфейс *Queue*. Он служит для создания очереди по приоритетам на основании компаратора очереди. Класс *PriorityQueue* является обобщенным и объявляется следующим образом:

```
class PriorityQueue<E>
```

где `E` обозначает тип объектов, которые будут храниться в очереди. Объект типа `PriorityQueue` представляет собой динамическую очередь, которая может при необходимости расширяться.

В классе `PriorityQueue` определяются следующие шесть конструкторов:

```
PriorityQueue()
PriorityQueue(int емкость)
PriorityQueue(int емкость, Comparator<? super E> компаратор)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)
```

Первый конструктор данного класса создает пустую очередь. Ее первоначальная емкость равна 11. Второй конструктор создает очередь с заданной начальной емкостью. Третий конструктор создает очередь заданной емкости с указанным компаратором. Последние три конструктора создают очереди, инициируемые элементами коллекций, задаваемых в качестве параметра `c`. Но в любом случае по мере ввода элементов в очередь ее емкость автоматически наращивается.

Если при построении очереди типа `PriorityQueue` компаратор не указан, то применяется компаратор, выбираемый по умолчанию для того типа данных, который сохраняется в очереди. Компаратор по умолчанию размещает элементы очереди по нарастающей. Таким образом, в начале (голове) очереди окажется элемент с наименьшим значением. Но, предоставив свой компаратор, можно задать другую схему сортировки элементов в очереди. Например, когда в очереди сохраняются элементы, содержащие метку времени, для этой очереди можно задать приоритеты таким образом, чтобы самые давние элементы располагались в начале очереди.

Вызвав метод `comparator()` из класса `PriorityQueue`, можно получить ссылку на компаратор, используемый в очереди, как показано ниже.

```
Comparator<? super E> comparator()
```

Этот метод возвращает компаратор. Если в данной очереди применяется естественный порядок сортировки, то возвращается пустое значение `null`. Следует, однако, иметь в виду, что порядок перебора элементов очереди типа `PriorityQueue` не определен, несмотря на то, что их можно перебрать, используя итератор. Чтобы правильно воспользоваться классом `PriorityQueue`, следует вызывать такие методы, как `offer()` и `poll()`, определенные в интерфейсе `Queue`.

Класс `ArrayDeque`

Класс `ArrayDeque` расширяет класс `AbstractCollection` и реализует интерфейс `Deque`. Он не добавляет свои методы. Класс `ArrayDeque` создает динамический массив, не имеющий ограничений по емкости. (Интерфейс `Deque` поддерживает реализации с ограниченной емкостью, но не накладывает на ее величину никаких ограничений.) Класс `ArrayDeque` является обобщенным и объявляется следующим образом:

```
class ArrayDeque<E>
```

590 Часть II. Библиотека Java

где **E** обозначает тип объекта, сохраняемого в коллекции. В классе **ArrayDeque** определяются следующие конструкторы:

```
ArrayDeque()
ArrayDeque(int размер)
ArrayDeque(Collection<? extends E> c)
```

Первый конструктор создает пустую двустороннюю очередь, первоначальная емкостью которой равна 16. Второй конструктор создает двустороннюю очередь указанной емкости. Третий конструктор создает двустороннюю очередь, инициализируемую заданной коллекцией *c*. Но в любом случае емкость увеличивается при вводе новых элементов в двустороннюю очередь по мере надобности.

В приведенном ниже примере программы демонстрируется применение класса **ArrayDeque** для организации стека.

```
// Продемонстрировать применения класса ArrayDeque
import java.util.*;

class ArrayDequeDemo {
    public static void main(String args[]) {
        // создать двустороннюю очередь
        ArrayDeque<String> adq = new ArrayDeque<String>();

        // использовать класс ArrayDeque для организации стека
        adq.push("A");
        adq.push("B");
        adq.push("D");
        adq.push("E");
        adq.push("F");

        System.out.print("Извлечение из стека: ");

        while(adq.peek() != null)
            System.out.print(adq.pop() + " ");

        System.out.println();
    }
}
```

Эта программа выводит следующий результат:

Извлечение из стека: F E D B A

Класс **EnumSet**

Класс **EnumSet** расширяет класс **AbstractSet** и реализует интерфейс **Set**. Он служит для создания множества, предназначенного для применения вместе с ключами перечислимого типа **enum**. Это обобщенный класс, объявляемый следующим образом:

```
class EnumSet<E extends Enum<E>>
```

где **E** обозначает элементы перечислимого типа. Следует иметь в виду, что класс **E** должен расширять класс **Enum<E>**, а это требует, чтобы элементы относились к указанному перечислимому типу **enum**.

В классе `EnumSet` конструкторы не определяются. Вместо этого для создания объектов используются фабричные методы, перечисленные в табл. 18.9. Обратите внимание на неоднократную перегрузку метода `of()`. Это делается из соображений эффективности. Передать известное количество аргументов, когда оно невелико, можно быстрее, чем делать это с помощью параметра переменной длины.

Таблица 18.9. Методы из класса `EnumSet`

Метод	Описание
<code>static <E extends Enum<E>></code> <code>EnumSet<E> allOf(Class<E> t)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы заданного перечисления <code>t</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> complementOf(EnumSet<E> e)</code>	Создает множество типа <code>EnumSet</code> , дополняющее элементы, отсутствующие в заданном множестве <code>e</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> copyOf(EnumSet<E> c)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы из заданного множества <code>c</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> copyOf(Collection<E> c)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы из заданной коллекции <code>c</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> noneOf(Class<E> t)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы, которые не входят в заданное перечисление <code>t</code> , которое по определению является пустым множеством
<code>static <E extends Enum<E>></code> <code>EnumSet<E> of (E v, E ... аргументы перечислений длины)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы <code>v</code> и нуль или дополнительные значения перечислимого типа
<code>static <E extends Enum<E>></code> <code>EnumSet<E> of (E v)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы <code>v</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> of (E v1, E v2)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы <code>v1</code> и <code>v2</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> of (E v1, E v2, E v3)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы от <code>v1</code> до <code>v3</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> of (E v1, E v2, E v3, E v4)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы от <code>v1</code> до <code>v4</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> of (E v1, E v2, E v3, E v4, E v5)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы от <code>v1</code> до <code>v5</code>
<code>static <E extends Enum<E>></code> <code>EnumSet<E> range (E начало, E конец)</code>	Создает множество типа <code>EnumSet</code> , содержащее элементы в заданных пределах от <code>начала</code> и до <code>конца</code>

ДОСТУП К КОЛЛЕКЦИЯМ ЧЕРЕЗ ИТЕРАТОР

Нередко требуется перебрать все элементы коллекции, например, вывести каждый ее элемент. Для этого можно, например, воспользоваться *итератором* – объектом класса, реализующего один из двух интерфейсов: `Iterator` или `ListIterator`. В частности, интерфейс `Iterator` позволяет организовать цикл для перебора коллекции, извлекая или удаляя из нее элементы. А интерфейс

`ListIterator` расширяет интерфейс `Iterator` для двустороннего обхода списка и видоизменения его элементов. Интерфейсы `Iterator` и `ListIterator` являются обобщенными и объявляются следующим образом:

```
interface Iterator<E>
interface ListIterator<E>
```

где `E` обозначает тип перебираемых объектов. В интерфейсе `Iterator` объявляются методы, перечисленные в табл. 18.10. А методы, объявляемые в интерфейсе `ListIterator`, перечислены в табл. 18.11. В обоих случаях операции, видоизменяющие базовую коллекцию, необязательны. Например, метод `remove()` генерирует исключение типа `UnsupportedOperationException`, если вызвать его для коллекции, доступной только для чтения. Возможны и другие исключения.

Таблица 18.10. Методы из интерфейса `Iterator`

Метод	Описание
<code>default void forEachRemaining(Consumer<? super E> действие)</code>	Выполняет заданное <i>действие</i> над каждым необработанным элементом коллекции (добавлен в версии JDK 8)
<code>boolean hasNext()</code>	Возвращает логическое значение <code>true</code> , если в коллекции еще имеются элементы, а иначе – логическое значение <code>false</code>
<code>E next()</code>	Возвращает следующий элемент из коллекции. Генерирует исключение типа <code>NoSuchElementException</code> , если в коллекции больше нет элементов
<code>void remove()</code>	Удаляет текущий элемент из коллекции. Генерирует исключение типа <code>IllegalStateException</code> , если предпринимается попытка вызвать метод <code>remove()</code> , которому не предшествовал вызов метода <code>next()</code> . В варианте этого метода по умолчанию генерируется исключение типа <code>UnsupportedOperationException</code>

Таблица 18.11. Методы из интерфейса `ListIterator`

Метод	Описание
<code>void add(E объект)</code>	Вводит заданный <i>объект</i> перед элементом, который должен быть возвращен в результате последующего вызова метода <code>next()</code>
<code>default void forEachRemaining(Consumer<? super E> действие)</code>	Выполняет заданное <i>действие</i> над каждым необработанным элементом коллекции (добавлен в версии JDK 8)
<code>boolean hasNext()</code>	Возвращает логическое значение <code>true</code> , если в списке имеется следующий элемент, а иначе – логическое значение <code>false</code>
<code>boolean hasPrevious()</code>	Возвращает логическое значение <code>true</code> , если в списке имеется предыдущий элемент, а иначе – логическое значение <code>false</code>

Окончание табл. 18.11

Метод	Описание
<code>E next()</code>	Возвращает следующий элемент из списка. Если следующий элемент отсутствует, то генерируется исключение типа <code>NoSuchElementException</code>
<code>int nextIndex()</code>	Возвращает индекс следующего элемента в списке. Если следующий элемент отсутствует, то возвращается длина списка
<code>E previous()</code>	Возвращает предыдущий элемент из списка. Если предыдущий элемент отсутствует, то генерируется исключение типа <code>NoSuchElementException</code>
<code>int previousIndex()</code>	Возвращает индекс предыдущего элемента в списке. Если предыдущий элемент отсутствует, то возвращается значение <code>-1</code>
<code>void remove()</code>	Удаляет текущий элемент из списка. Если метод <code>remove()</code> вызывается до метода <code>next()</code> или <code>previous()</code> , то генерируется исключение типа <code>IllegalStateException</code>
<code>void set(E объект)</code>	Присваивает заданный <code>объект</code> текущему элементу списка. Это элемент, возвращаемый в результате последнего вызова метода <code>next()</code> или <code>previous()</code>

На заметку! В версии JDK 8 появилась возможность для циклического обхода коллекции средствами интерфейса `Splitterator`. Данный интерфейс действует иначе, чем интерфейс `Iterator`, и будет описан далее.

Применение интерфейса `Iterator`

Прежде чем обратиться к коллекции через итератор, следует получить его. В каждом классе коллекций предоставляется метод `iterator()`, возвращающий итератор на начало коллекции. Используя объект итератора, можно получить доступ к каждому элементу коллекции по очереди. В общем, применение итератора для перебора содержимого коллекции сводится к выполнению следующих действий.

- Установить итератор на начало коллекции, получив его из метода `iterator()`, вызываемого для коллекции.
- Организовать цикл, в котором вызывается метод `hasNext()`. Перебирать содержимое коллекции до тех пор, пока метод `hasNext()` не возвратит логическое значение `true`.
- Получить в цикле каждый элемент коллекции, вызывая метод `next()`.

Что касается видов коллекций, реализующих интерфейс `List`, то получить для них итератор можно, вызывая метод `ListIterator()`. Как пояснялось ранее, итератор списка обеспечивает доступ к элементам такой коллекции, как в прямом, так и в обратном направлении, а также позволяет видоизменять элементы списка. А в остальном интерфейс `ListIterator` применяется таким же образом, как и интерфейс `Iterator`.

594 Часть II. Библиотека Java

В следующем примере программы выполняются все перечисленные выше действия и демонстрируется применение обоих интерфейсов, `Iterator` и `ListIterator`. В данном примере в качестве перебираемой коллекции используется объект типа `ArrayList`, но общие принципы перебора содержимого с помощью итераторов применимы к коллекциям любого вида. Безусловно, интерфейс `ListIterator` доступен только тем видам коллекций, которые реализуют интерфейс `List`.

```
// Продемонстрировать применение итераторов
import java.util.*;

class IteratorDemo {
    public static void main(String args[]) {
        // создать списочный массив
        ArrayList<String> al = new ArrayList<String>();

        // ввести элементы в списочный массив
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");

        // использовать итераторы для вывода содержимого
        // списочного массива al
        System.out.print(
            "Исходное содержимое списочного массива al: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();

        //видоизменить перебираемые объекты
        ListIterator<String> litr = al.listIterator();
        while(litr.hasNext()) {
            String element = litr.next();
            litr.set(element + "+");
        }

        System.out.print(
            "Измененное содержимое списочного массива al: ");
        itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();

        // а теперь отобразить список в обратном порядке
        System.out.print("Измененный в обратном порядке список: ");
        while(litr.hasPrevious()) {
            String element = litr.previous();
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Ниже приведен результат, выводимый данной программой.

Исходное содержимое списочного массива `al`: C A E B D F
 Модифицированное содержимое списочного массива `al`: C+ A+ E+ B+ D+ F+
 Модифицированный в обратном порядке список: F+ D+ B+ E+ A+ C+

Обратите особое внимание на вывод списка в обратном порядке. После видоизменения списка итератор `litr` указывает на конец списка. (Напомним, что метод `litr.hasNext()` возвращает логическое значение `false`, если достигнут конец списка.) Для перебора списка в обратном порядке в данной программе по-прежнему используется итератор `litr`, но на этот раз в ней проверяется, существует ли предыдущий элемент. И до тех пор, пока это делается, выводится каждый элемент, получаемый из списка.

Цикл `for` в стиле `for each` как альтернатива итераторам

Если не требуется видоизменять содержимое коллекции или извлекать из нее элементы в обратном порядке, в таком случае цикл `for` в стиле `for each` может оказаться более удобной альтернативой итераторам. Напомним, что в цикле `for` можно перебирать любую коллекцию объектов, реализующую интерфейс `Iterable`. А поскольку все классы коллекций реализуют этот интерфейс, то ими можно оперировать в цикле `for`.

В следующем примере программы цикл `for` в стиле `for each` используется для суммирования содержимого коллекции:

```
// Применение цикла for в стиле for each
// для перебора элементов коллекции
import java.util.*;

class ForEachDemo {
    public static void main(String args[]) {
        // создать списочный массив для целых чисел
        ArrayList<Integer> vals = new ArrayList<Integer>();

        // ввести числовые значения в списочный массив
        vals.add(1);
        vals.add(2);
        vals.add(3);
        vals.add(4);
        vals.add(5);

        // организовать цикл для вывода числовых значений
        System.out.print(
            "Исходное содержимое списочного массива vals: ");
        for(int v : vals)
            System.out.print(v + " ");

        System.out.println();

        // суммировать числовые значения в цикле for
        int sum = 0;
        for(int v : vals)
            sum += v;

        System.out.println("Сумма числовых значений: " + sum);
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Исходное содержимое списочного массива vals: 1 3 4 5
Сумма числовых значений: 15
```

Как видите, организовать цикл `for` значительно проще и короче, чем использовать итератор. Но он подходит для перебора элементов коллекции только в прямом направлении и не позволяет видоизменять элементы коллекции.

Итераторы-разделители

В версии JDK 8 внедрен новый тип итератора, называемый *итератором-разделителем* и определяемый в интерфейсе `Spliterator`. Итераторы-разделители позволяют перебирать последовательность элементов, и в этом отношении они подобны описанным выше итераторам. Но применяются они иначе. Кроме того, в интерфейсе `Spliterator` предоставляются значительно более широкие функциональные возможности, чем в интерфейсе `Iterator` или `ListIterator`. Вероятно, наиболее важной особенностью интерфейса `Spliterator` является его способность поддерживать параллельную итерацию отдельных частей последовательности элементов, а следовательно, и параллельное программирование, подробнее рассматриваемое в главе 28. По интерфейсу `Spliterator` можно применять и в том случае, когда распараллеливание выполняемых операций не требуется. Одной из причин для такого применения интерфейса `Spliterator` может служить то обстоятельство, что в одном его методе сочетаются операции, выполняемые методами `hasNext()` и `next()` над перебираемыми элементами.

Интерфейс `Spliterator` является обобщенным и объявляется следующим образом:

```
interface Spliterator<T>
```

где `T` обозначает тип перебираемых элементов. В интерфейсе `Spliterator` объявляются методы, перечисленные в табл. 18.12.

Таблица 18.12. Методы из интерфейса `Spliterator`

Метод	Описание
<code>int characteristics()</code>	Возвращает характеристики вызывающего итератора-разделителя, представленные в виде целочисленного значения
<code>long estimateSize()</code>	Оценивает количество элементов, которое остались перебрать, и возвращает полученный результат. Если это количество нельзя получить по какой-нибудь причине, то возвращается значение константы <code>Long.MAX_VALUE</code>
<code>default void forEachRemaining(Consumer<? super T> действие)</code>	Выполняет заданное <i>действие</i> над каждым необработанным элементом в источнике данных

Окончание табл. 18.12

Метод	Описание
<code>default Comparator<? super T> getComparator()</code>	Возвращает компаратор, используемый вызывающим итератором-разделителем, или пустое значение <code>null</code> , если используется естественное упорядочение. А если последовательность не упорядочена, то генерируется исключение типа <code>IllegalStateException</code>
<code>default long getExactSizeIfKnown()</code>	Если установлен размер вызывающего итератора-разделителя, то возвращается количество элементов, которое осталось перебрать, а иначе – значение <code>-1</code>
<code>default boolean hasCharacteristics(int val)</code>	Возвращает логическое значение <code>true</code> , если у вызывающего итератора-разделителя имеются характеристики, передаваемые в качестве параметра <code>val</code> , а иначе – логическое значение <code>false</code>
<code>boolean tryAdvance(Consumer<? super T> действие)</code>	Выполняет заданное <i>действие</i> над следующим элементом в итерации. Возвращает логическое значение <code>true</code> , если следующий элемент присутствует, а иначе – логическое значение <code>false</code>
<code>Spliterator<T> trySplit()</code>	Разделяет, если это возможно, вызывающий итератор-разделитель, возвращая ссылку на новый итератор-разделитель для последующего разделения, а иначе – пустое значение <code>null</code> . При удачном исходе разделения исходный итератор-разделитель перебирает одну часть последовательности, а возвращаемый итератор-разделитель – остальную ее часть

Интерфейс `Spliterator` применяется для решения основных задач итерации очень просто. Для этого достаточно вызывать метод `tryAdvance()` до тех пор, пока он не возвратит логическое значение `false`. Если же требуется выполнить одно и то же действие над каждым элементом последовательности, то для этой цели имеется более простая альтернатива – вызвать метод `forEachRemaining()`. В обоих случаях действие, происходящее на каждом шаге итерации, определяется тем, что именно объект типа `Consumer` собирается делать с каждым элементом, где `Consumer` – это функциональный интерфейс, выполняющий действие над объектом. Этот обобщенный функциональный интерфейс определен в пакете `java.util.function`, подробно рассматриваемом в главе 19. В функциональном интерфейсе `Consumer` определяется единственный абстрактный метод `accept()`, общая форма которого приведена ниже.

```
void accept(T ссылка_на_объект)
```

Если на каждом шаге итерации вызывается метод `tryAdvance()`, то следующий элемент последовательности передается по ссылке, обозначаемой параметром `ссылка_на_объект`. Зачастую интерфейс `Consumer` проще всего реализовать с помощью лямбда-выражения.

598 Часть II. Библиотека Java

В приведенной ниже программе демонстрируется простой пример применения интерфейса `Spliterator`. В этой программе демонстрируется также применение обоих методов, `tryAdvance()` и `forEachRemaining()`. Обратите внимание на то, что эти методы сочетают в одном своем вызове действия методов `next()` и `hasNext()` из интерфейса `Iterator`.

```
// Продемонстрировать простое применение интерфейса Spliterator
import java.util.*;

class SpliteratorDemo {

    public static void main(String args[]) {
        // создать списочный массив числовых значений типа double
        ArrayList<Double> vals = new ArrayList<>();

        // ввести значения в списочный массив
        vals.add(1.0);
        vals.add(2.0);
        vals.add(3.0);
        vals.add(4.0);
        vals.add(5.0);

        // вызвать метод tryAdvance() для вывода содержимого
        // списочного массива vals
        System.out.print("Содержимое списочного массива vals:\n");
        Spliterator<Double> splitr = valsspliterator();
        while(splitr.tryAdvance((n) -> System.out.println(n)));
        System.out.println();

        // создать новый списочный массив, содержащий квадратные
        // корни числовых значений из списочного массива vals
        splitr = valsspliterator();
        ArrayList<Double> sqrs = new ArrayList<>();
        while(splitr.tryAdvance((n) -> sqrs.add(Math.sqrt(n))));

        // вызвать метод forEachRemaining() для вывода содержимого
        // списочного массива sqrss
        System.out.print("Содержимое списочного массива sqrs:\n");
        splitr = sqrsspliterator();
        splitr.forEachRemaining((n) -> System.out.println(n));
        System.out.println();
    }
}
```

Ниже приведен результат, выводимый данной программой.

Содержимое списочного массива vals:

1.0
2.0
3.0
4.0
5.0

Содержимое списочного массива sqrs:

1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979

Несмотря на то что в данной программе демонстрируется механизм применения интерфейса `Spliterator`, она не раскрывает весь его потенциал. Как упоми-

налось выше, наибольшую выгоду применение интерфейса `Spliterator` приносит в тех случаях, когда требуется параллельная обработка.

Обратите внимание на методы `characteristics()` и `hasCharacteristics()`, перечисленные в табл. 18.12. У каждого итератора-разделителя типа `Spliterator` имеются свойства, называемые *характеристиками*. Эти характеристики определяются в статических полях типа `int` интерфейса `Spliterator`, в том числе `SORTED`, `DISTINCT`, `SIZED` и `IMMUTABLE` и прочих полях. Для получения характеристик итератора-разделителя достаточно вызвать метод `characteristics()`, а для выявления отдельной характеристики у итератора-разделителя — метод `hasCharacteristics()`. Зачастую получать характеристики итератора-разделителя не требуется, но иногда они помогают в написании эффективного, устойчивого кода.

На заметку! Рассмотрение интерфейса `Spliterator` будет продолжено в главе 29, где обсуждается его применение в контексте нового прикладного программного интерфейса API потоков данных. Лямбда-выражения рассматриваются в главе 15, а распараллеливание и параллельное программирование — в главе 28.

Несколько подчиненных интерфейсов, вложенных в интерфейс `Spliterator`, предназначены для применения вместе с примитивными типами данных `double`, `int` и `long`. Это интерфейсы `Spliterator.OfDouble`, `Spliterator.OfInt` и `Spliterator.OfLong`. Имеется также обобщенный вариант интерфейса `Spliterator.OfPrimitive`, предоставляющий дополнительные удобства и подчиненный упомянутым выше интерфейсам.

Сохранение объектов пользовательских классов в коллекциях

Ради простоты во всех приведенных ранее примерах программ в коллекциях сохранялись объекты встроенных классов наподобие `String` или `Integer`. Безусловно, сохранение в коллекции не ограничивается только встроенными объектами встроенных классов. Напротив, эффективность коллекций в том и состоит, что в них можно хранить любой тип объектов, включая объекты тех классов, которые вы создаете сами. Рассмотрим в качестве примера следующую программу, где класс `LinkedList` используется для сохранения почтовых адресов:

```
// Простой пример обработки списка почтовых адресов
import java.util.*;

class Address {
    private String name;
    private String street;
    private String city;
    private String state;
    private String code;
    Address(String n, String s, String c,
            String st, String cd) {
        name = n;
        street = s;
        city = c;
```

600 Часть II. Библиотека Java

```
state = st;
code = cd;
}

public String toString() {
    return name + "\n" + street + "\n" +
    city + " " + state + " " + code;
}
}

class MailList {
    public static void main(String args[]) {
        LinkedList<Address> ml = new LinkedList<Address>();

        // ввести элементы в связный список
        ml.add(new Address("J.W. West", "11 Oak Ave",
                           "Urbana", "IL", "61801"));
        ml.add(new Address("Ralph Baker", "1142 Maple Lane",
                           "Mahomet", "IL", "61853"));
        ml.add(new Address("Tom Carlton", "867 Elm St",
                           "Champaign", "IL", "61820"));

        // вывести список почтовых адресов
        for(Address element : ml)
            System.out.println(element + "\n");

        System.out.println();
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
J.W. West
11 Oak Ave
Urbana IL 61801

Ralph Baker
1142 Maple Lane
Mahomet IL 61853

Tom Carlton
867 Elm St
Champaign IL 61820
```

Помимо сохранения объектов пользовательских классов в коллекции, данная программа заслуживает внимания еще и потому, что она довольно короткая. Если принять во внимание, что для сохранения, извлечения и обработки почтовых адресов понадобилось всего около 50 строк кода, то станет очевидной эффективность каркаса коллекций. Ведь если запрограммировать все эти функциональные возможности вручную, то программа окажется в несколько раз длиннее. Коллекции предлагают готовые решения для широкого круга задач программирования. Их следует использовать всякий раз, когда позволяет ситуация.

Интерфейс RandomAccess

Этот интерфейс не содержит ни одного члена. Но, реализуя его, коллекция извещает о том, что она поддерживает эффективный произвольный доступ к своим

элементам. Даже если в коллекции и поддерживается произвольный доступ к ее элементам, такой доступ может оказаться недостаточно эффективным. Проверяя интерфейс `RandomAccess` во время выполнения, в прикладном коде можно выяснить, допускает ли конкретная коллекция некоторые виды операций произвольного доступа, и, в частности, насколько они применимы к крупным коллекциям. (Чтобы определить, реализует ли класс коллекции интерфейс `RandomAccess`, можно воспользоваться оператором `instanceof`.) Интерфейс `RandomAccess` реализуется в классе `ArrayList` и, среди прочего, в унаследованном классе `Vector`.

Обращение с отображениями

Отображение представляет собой объект, сохраняющий связи между ключами и значениями в виде пар “ключ–значение”. По заданному ключу можно найти его значение. Ключи и значения являются объектами. Ключи могут быть однозначными, а значения – дублированными. В одних отображениях допускаются пустые ключи и пустые значения, а в других – они не допускаются.

В отношении отображений необходимо иметь в виду следующее: они не реализуют интерфейс `Iterable`. Это означает, что перебрать содержимое отображения, организовав цикл `for` в стиле `for each, не удастся`. Более того, нельзя получить итератор отображения. Но, как будет показано ниже, можно получить представление отображения в виде коллекции, которое допускает перебор содержимого в цикле или с помощью итератора.

Интерфейсы отображений

Интерфейсы отображений определяют их характер и особенности, поэтому начать рассмотрение отображений следует с их интерфейсов. Отображения поддерживаются в интерфейсах, перечисленных в табл. 18.13. Каждый из этих интерфейсов рассматривается далее по отдельности.

Таблица 18.13. Интерфейсы, поддерживающие отображения

Интерфейс	Описание
<code>Map</code>	Отображает однозначные ключи на значения
<code>Map.Entry</code>	Описывает элемент отображения (пару “ключ–значение”). Это внутренний класс интерфейса <code>Map</code>
<code>NavigableMap</code>	Расширяет интерфейс <code>SortedMap</code> для извлечения элементов из отображения по критерию поиска наиболее точного совпадения
<code>SortedMap</code>	Расширяет интерфейс <code>Map</code> таким образом, чтобы ключи располагались по параболической

Интерфейс `Map`

Интерфейс `Map` отображает однозначные ключи на значения. *Ключ* – это объект, используемый для последующего извлечения данных. Задавая ключ и значение, можно размещать значение в отображении, представленном объектом типа

Map. Сохранив значение по ключу, можно получить его обратно по этому же ключу. Интерфейс Map является обобщенным и объявляется приведенным ниже образом, где K обозначает тип ключей, а V – тип хранимых в отображении значений.

```
interface Map<K, V>
```

Методы, объявляемые в интерфейсе Map, перечислены в табл. 18.14. Некоторые из них генерируют исключение типа ClassCastException, если заданный объект несовместим с элементами отображения. Исключение типа NullPointerException генерируется при попытке использовать пустой объект, когда данное отображение этого не допускает. А исключение типа UnsupportedOperationException генерируется при попытке изменить неизменяемое отображение.

Таблица 18.14. Методы из интерфейса Map

Метод	Описание
<code>default V compute(K k, BiFunction<? super K, ? super V, ? extends V> функция)</code>	Вызывает заданную <i>функцию</i> для построения нового значения. Если <i>функция</i> возвращает непустое значение, то в отображение вводится новая пара “ключ–значение”, удаляется любая ранее существовавшая пара и возвращается новое значение. А если <i>функция</i> возвращает пустое значение <code>null</code> , то удаляется любая ранее существовавшая пара и возвращается пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>default V computeIfAbsent(K k, Function<? super K, ? extends V> функция)</code>	Возвращает значение, связанное с указанным ключом <i>k</i> . В противном случае создается новое значение, для чего вызывается заданная <i>функция</i> , в отображение вводится новая пара “ключ–значение” и возвращается созданное значение. Если же новое значение создать нельзя, то возвращается пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>default V computeIfPresent(K k, BiFunction<? super K, ? super V, ? extends V> функция)</code>	Если в отображении присутствует указанный ключ <i>k</i> , то для создания нового значения вызывается заданная <i>функция</i> и новое значение заменяет прежнее в отображении. В этом случае возвращается новое значение. Если же заданная <i>функция</i> возвращает пустое значение <code>null</code> , то из отображения удаляются существующие в нем ключ и значение и затем возвращается пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>void clear()</code>	Удаляет все пары “ключ–значение” из вызывающего отображения
<code>boolean containsKey(Object k)</code>	Возвращает логическое значение <code>true</code> , если вызывающее отображение содержит указанный ключ <i>k</i> , а иначе – логическое значение <code>false</code>
<code>boolean containsValue(Object v)</code>	Возвращает логическое значение <code>true</code> , если вызывающее отображение содержит значение <i>v</i> , а иначе – логическое значение <code>false</code>

Продолжение табл. 18.14

Метод	Описание
<code>Set<Map.Entry<K, V>> entrySet()</code>	Возвращает множество типа <code>Set</code> , содержащее все записи из вызывающего отображения в виде объектов типа <code>Map.Entry</code> . Следовательно, этот метод возвращает представление вызывающего отображения в виде множества
<code>boolean equals(Object объект)</code>	Возвращает логическое значение <code>true</code> , если заданный <code>объект</code> является отображением типа <code>Map</code> , содержащим одинаковые значения, а иначе — логическое значение <code>false</code>
<code>default void forEach(BiConsumer<? super K, ? super V> действие)</code>	Выполняет заданное <code>действие</code> над каждым элементом вызывающего отображения. Если в ходе этого процесса удаляется элемент, то генерируется исключение типа <code>ConcurrentModificationException</code> (добавлен в версии JDK 8)
<code>V get(Object k)</code>	Возвращает значение, связанное с указанным ключом <code>k</code> . Если же ключ не найден, то возвращается пустое значение <code>null</code>
<code>default V getOrDefault(Object k, V заданное_значение)</code>	Возвращает значение, связанное с указанным ключом <code>k</code> , если оно присутствует в вызывающем отображении, а иначе — <code>заданное_значение</code> (добавлен в версии JDK 8)
<code>int hashCode()</code>	Возвращает хеш-код вызывающего отображения
<code>boolean isEmpty()</code>	Возвращает логическое значение <code>true</code> , если вызывающее отображение пусто, а иначе — логическое значение <code>false</code>
<code>Set<K> keySet()</code>	Возвращает множество, содержащее ключи из вызывающего отображения. Следовательно, этот метод возвращает представление ключей в вызывающем отображении в виде множества
<code>default V merge(K k, V v, BiFunction<? super V, ? super V> функция)</code>	Если в вызывающем отображении отсутствует указанный ключ <code>k</code> , то в него вводится пара “ключ-значение”, определяемая параметрами <code>k, v</code> , а затем возвращается значение <code>v</code> . В противном случае заданная <code>функция</code> возвращает новое значение, исходя из прежнего значения и ключ обновляется для доступа к этому значению, а затем оно возвращается из метода <code>merge()</code> . Если же заданная <code>функция</code> возвращает пустое значение <code>null</code> , то ключ и значения, существующие в вызывающем отображении, удаляются из него и затем возвращается пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>V put(K k, V v)</code>	Вводит новое значение <code>v</code> в вызывающее отображение, перезаписывая любое предшествующее значение, связанное с заданным ключом <code>k</code> . Возвращает пустое значение <code>null</code> , если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом

Окончание табл. 18.14

Метод	Описание
<code>void putAll(Map<? extends K, ? extends V> m)</code>	Вводит все записи из заданного отображения <i>m</i> в вызывающее отображение
<code>default V putIfAbsent(K k, V v)</code>	Вводит пару “ключ–значение”, определяемую параметрами <i>k</i> , <i>v</i> , в вызывающее отображение, если она отсутствует в нем или же если значение, связанное с заданным ключом <i>k</i> , оказывается пустым. В этом случае возвращает пустое значение <code>null</code> , а иначе – прежнее значение (добавлен в версии JDK 8)
<code>V remove(Object k)</code>	Удаляет запись по заданному ключу <i>k</i>
<code>default boolean remove(Object k, Object v)</code>	Если пара “ключ–значение”, определяемая параметрами <i>k</i> , <i>v</i> , присутствует в вызывающем отображении, то она удаляется и затем возвращается логическое значение <code>true</code> , а иначе – логическое значение <code>false</code> (добавлен в версии JDK 8)
<code>default boolean replace(K k, V <i>прежнее_значение</i>, V <i>новое_значение</i>)</code>	Если пара “ключ–значение”, определяемая параметрами <i>v</i> и <i>прежнее_значение</i> , присутствует в вызывающем отображении, то это значение заменяется на <i>новое_значение</i> и затем возвращается логическое значение <code>true</code> , а иначе – логическое значение <code>false</code> (добавлен в версии JDK 8)
<code>default V replace(K k, V v)</code>	Если в вызывающем отображении присутствует заданный ключ <i>k</i> , то значение по этому ключу заменяется новым значением <i>v</i> и возвращается прежнее значение, а иначе – пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>default void replaceAll(BiFunction<? super K, ? super V, ? extends V> <i>функция</i>)</code>	Выполняет заданную <i>функцию</i> для каждого элемента в вызывающем отображении, заменяя элемент результатом, возвращаемым заданной <i>функцией</i> . Если в ходе этого процесса удаляется элемент, то генерируется исключение типа <code>ConcurrentModificationException</code> (добавлен в версии JDK 8)
<code>int size()</code>	Возвращает количество пар “ключ–значение” в вызывающем отображении
<code>Collection<V> values()</code>	Возвращает коллекцию, содержащую значения из вызывающего отображения. Следовательно, этот метод возвращает представление значений в вызывающем отображении в виде коллекции

Обращение с отображениями опирается на две основные операции, выполняемые методами `get()` и `put()`. Чтобы ввести значение в отображение, следует вызвать метод `put()`, указав ключ и значение, а для того чтобы получить значение из отображения – вызвать метод `get()`, передав ему ключ в качестве аргумента. Поэтому ключу будет возвращено связанное с ним значение.

Как упоминалось ранее, отображения не реализуют интерфейс `Collection`, хотя являются частью каркаса коллекций. Тем не менее можно получить представ-

ление отображения в виде коллекции. Для этого можно воспользоваться методом `entrySet()`, возвращающим множество, содержащее элементы отображения. Чтобы получить представление ключей в отображении в виде коллекции, следует вызвать метод `keySet()`, а для того чтобы получить представление значений в отображении в виде коллекции — метод `values()`. Все три представления отображений и их элементов в виде коллекций относятся к тем средствам, с помощью которых отображения интегрируются в крупный каркас коллекций.

Интерфейс `SortedMap`

Этот интерфейс расширяет интерфейс `Map`. Он обеспечивает размещение записей в отображении по порядку нарастания ключей. Интерфейс `SortedMap` является обобщенным и объявляется приведенным ниже образом, где `K` обозначает тип ключей, а `V` — тип хранимых в отображении значений.

```
interface SortedMap<K, V>
```

Методы, объявляемые в интерфейсе `SortedMap`, перечислены в табл. 18.15. Некоторые из них генерируют исключение типа `NoSuchElementException`, если вызывающее отображение пусто. Исключение типа `ClassCastException` генерируется в том случае, если заданный объект несовместим с элементами, хранящимися в отображении. Исключение типа `NullPointerException` генерируется при попытке использовать пустой объект, когда пустые объекты в данном отображении не допускаются. А исключение типа `IllegalArgumentException` генерируется в том случае, если указан неверный аргумент.

Таблица 18.15. Методы из интерфейса `SortedMap`

Метод	Значение
<code>Comparator<? super K> comparator()</code>	Возвращает компаратор вызывающего отсортированного отображения. Если в отображении применяется естественное упорядочение элементов, то возвращается пустое значение <code>null</code>
<code>K firstKey()</code>	Возвращает первый ключ из вызывающего отображения
<code>SortedMap<K, V> headMap(K конец)</code>	Возвращает отсортированное отображение, содержащее те элементы из вызывающего отображения, ключи которых меньше, чем указанный <code>конец</code>
<code>K lastKey()</code>	Возвращает последний ключ в вызывающем отображении
<code>SortedMap<K, V> subMap(K начало, K конец)</code>	Возвращает отображение, содержащее элементы вызывающего отображения, ключ которых больше, чем <code>начало</code> , или равен ему и меньше, чем <code>конец</code>
<code>SortedMap<K, V> tailMap(K начало)</code>	Возвращает отсортированное отображение, содержащее те элементы вызывающего отображения, ключ которых больше, чем указанное <code>начало</code>

Отсортированные отображения обеспечивают очень эффективное манипулирование подотображениями (иными словами, подмножествами отображений). Для получения подотображений служат методы `headMap()`, `tailMap()` или `subMap()`. Подотображение, возвращаемое этими методами, поддерживается вызывающим

отображением. При изменении одного изменяется другое. Для получения первого ключа из подмножества отображения следует вызвать метод `firstKey()`, а для получения последнего ключа — метод `lastKey()`.

Интерфейс NavigableMap

Интерфейс `NavigableMap` расширяет интерфейс `SortedMap` и определяет поведение отображения, поддерживающего извлечение записей из него по наиболее точному совпадению с заданным ключом или несколькими ключами. Интерфейс `NavigableMap` является обобщенным и объявляется следующим образом:

```
interface NavigableMap<K, V>
```

где `K` обозначает тип ключей, а `V` — тип значений, связанных с ключами. Помимо методов, наследуемых из интерфейса `SortedMap`, в интерфейс `NavigableMap` введены методы, перечисленные в табл. 18.16. Некоторые из них генерируют исключение типа `ClassCastException`, если объект несовместим с ключами отображения. Исключение типа `NullPointerException` генерируется при попытке использовать пустой объект, когда пустые ключи в отображении не допускаются. А исключение типа `IllegalArgumentException` передается при указании неверного аргумента.

Таблица 18.16. Методы из интерфейса NavigableMap

Метод	Значение
<code>Map.Entry<K, V> ceilingEntry(K объект)</code>	Выполняет поиск в отображении наименьшего ключа <code>k</code> по критерию <code>k >= объект</code> . Если такой ключ найден, то возвращается запись по этому ключу, а иначе — пустое значение <code>null</code>
<code>K ceilingKey(K объект)</code>	Выполняет поиск в отображении наименьшего ключа <code>k</code> по критерию <code>k >= объект</code> . Если такой ключ найден, то он возвращается, а иначе — пустое значение <code>null</code>
<code>NavigableSet<K> descendingKeySet()</code>	Возвращает множество типа <code>NavigableSet</code> , содержащее ключи в вызывающем отображении в обратном порядке. Следовательно, этот метод возвращает обратное представление ключей в отображении в виде множества. Результирующее множество опирается на вызывающее отображение
<code>NavigableMap<K, V> descendingMap()</code>	Возвращает множество типа <code>NavigableSet</code> , обратное вызывающему отображению. Результирующее множество опирается на вызывающее отображение
<code>Map.Entry<K, V> firstEntry()</code>	Возвращает первую запись в отображении. Это запись с наименьшим ключом
<code>Map.Entry<K, V> floorEntry(K объект)</code>	Выполняет поиск в отображении наибольшего ключа <code>k</code> по критерию <code>k <= объект</code> . Если такой ключ найден, то возвращается запись по этому ключу, а иначе — пустое значение <code>null</code>
<code>K floorKey(K объект)</code>	Выполняет поиск в отображении наибольшего ключа <code>k</code> по критерию <code>k <= объект</code> . Если такой ключ найден, то он возвращается, а иначе — пустое значение <code>null</code>

Продолжение табл. 18.16

Метод	Значение
<code>NavigableMap<K, V> headMap (K верхняя_граница, boolean включительно)</code>	Возвращает множество типа <code>NavigableSet</code> , содержащее все записи из вызывающего отображения по ключам меньше, чем заданная <i>верхняя_граница</i> . Если параметр <i>включительно</i> принимает логическое значение <code>true</code> , то в результирующее множество включается элемент, равный заданной <i>верхней_границе</i> . Результирующее множество опирается на вызывающее отображение
<code>Map.Entry<K, V> higherEntry (K объект)</code>	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k > объект</i> . Если такой ключ найден, то возвращается запись по этому ключу, а иначе – пустое значение <code>null</code>
<code>K higherKey (K объект)</code>	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k > объект</i> . Если такой ключ найден, то он возвращается, а иначе – пустое значение <code>null</code>
<code>Map.Entry<K, V> lastEntry ()</code>	Возвращает последнюю запись в отображении. Это запись с наибольшим ключом
<code>Map.Entry<K, V> lowerEntry (K объект)</code>	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k < объект</i> . Если такой ключ найден, то возвращается запись по этому ключу, а иначе – пустое значение <code>null</code>
<code>K lowerKey (K объект)</code>	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k < объект</i> . Если такой ключ найден, то он возвращается, а иначе – пустое значение <code>null</code>
<code>NavigableSet<K> navigableKeySet ()</code>	Возвращает множество типа <code>NavigableSet</code> , содержащее ключи из вызывающего отображения. Результирующее множество опирается на вызывающее отображение
<code>Map.Entry<K, V> pollFirstEntry ()</code>	Возвращает первую запись в отображении, попутно удаляя ее. Это запись по наименьшему ключу, поскольку отображение отсортировано. Если же отображение оказывается пустым, то возвращается пустое значение <code>null</code>
<code>Map.Entry<K, V> pollLastEntry ()</code>	Возвращает последнюю запись в отображении, попутно удаляя ее. Это запись по наибольшему ключу, поскольку отображение отсортировано. Если же отображение оказывается пустым, то возвращается пустое значение <code>null</code>
<code>NavigableMap<K, V> subMap (K нижняя_граница, boolean включая_нижнюю_границу, K верхняя_граница, boolean включая_верхнюю_границу)</code>	Возвращает отображение типа <code>NavigableSet</code> , содержащее все записи из вызывающего отображения по ключам меньше, чем <i>верхняя_граница</i> , и больше, чем <i>нижняя_граница</i> . Если параметр <i>включая_нижнюю_границу</i> принимает логическое значение <code>true</code> , то в результирующее отображение включается элемент, равный заданной <i>нижней_границе</i> . А если параметр <i>включая_верхнюю_границу</i> принимает логическое значение <code>true</code> , то в результирующее отображение включается элемент, равный заданной <i>верхней_границе</i> . Результирующее отображение опирается на вызывающее отображение

Окончание табл. 18.16

Метод	Значение
<code>NavigableMap<K, V> tailMap(K нижняя_граница, boolean включительно)</code>	Возвращает отображение типа <code>NavigableSet</code> , содержащее все записи из вызывающего отображения по ключам больше, чем заданная <i>нижняя_граница</i> . Если параметр <i>включительно</i> принимает логическое значение <code>true</code> , то в результирующее отображение включается элемент, равный заданной <i>нижней_границе</i> . Результирующее отображение опирается на вызывающее отображение

Интерфейс `Map.Entry`

Этот интерфейс позволяет обращаться с отдельными записями в отображении. Напомним, что метод `entrySet()`, объявляемый в интерфейсе `Map`, возвращает множество типа `Set`, содержащее записи из отображения. Каждый элемент этого множества представляет собой объект типа `Map.Entry`. Интерфейс `Map.Entry` является обобщенным и объявляется следующим образом:

```
interface Map.Entry<K, V>
```

где `K` обозначает тип ключей, а `V` – тип хранимых в отображении значений. В табл. 18.17 перечислены нестатические методы, объявляемые в интерфейсе `Map.Entry`. В версии JDK 8 в интерфейс введены два статических метода. Первый из них называется `comparingByKey()` и возвращает компаратор типа `Comparator`, сравнивающий записи в отображении по заданному ключу. А второй называется `comparingByValue()` и возвращает компаратор типа `Comparator`, сравнивающий записи в отображении по указанному значению.

Таблица 18.17. Методы из интерфейса `Map.Entry`

Метод	Значение
<code>boolean equals (Object объект)</code>	Возвращает логическое значение <code>true</code> , если заданный <i>объект</i> представляет запись из отображения типа <code>Map.Entry</code> , ключ и значение в которой такие же, как и у вызывающего объекта
<code>K getKey()</code>	Возвращает ключ данной записи из отображения
<code>V getValue()</code>	Возвращает значение данной записи из отображения
<code>int hashCode()</code>	Возвращает хеш-код данной записи из отображения
<code>V setValue (V v)</code>	Устанавливает указанное значение <code>v</code> в данной записи из отображения. Если значение <code>v</code> не относится к типу, допустимому для данного отображения, то генерируется исключение типа <code>ClassCastException</code> . Если же значение <code>v</code> указано неверно, то генерируется исключение типа <code>IllegalArgumentException</code> . А если значение <code>v</code> оказывается пустым (<code>null</code>) и в отображении нельзя хранить пустые ключи, то генерируется исключение типа <code>NullPointerException</code> . И наконец, если в отображение нельзя вносить изменения, то генерируется исключение типа <code>UnsupportedOperationException</code>

Классы отображений

Интерфейсы отображений реализуются в нескольких классах. Классы, которые могут быть использованы для отображений, перечислены в табл. 18.18. Следует иметь в виду, что класс `AbstractMap` служит суперклассом для всех конкретных реализаций отображений.

Класс `WeakHashMap` реализует отображение, в котором используются так называемые “слабые ключи”, что позволяет собирать в “мусор” запись из отображения как ненужный объект, когда ее ключ больше не используется. Этот класс подробно здесь не обсуждается, а прочие классы отображений описываются далее.

Таблица 18.18. Классы отображений

Класс	Описание
<code>AbstractMap</code>	Реализует большую часть интерфейса <code>Map</code>
<code>EnumMap</code>	Расширяет класс <code>AbstractMap</code> для применения вместе с ключами типа <code>enum</code>
<code>HashMap</code>	Расширяет класс <code>AbstractMap</code> для применения хеш-таблицы
<code>TreeMap</code>	Расширяет класс <code>AbstractMap</code> для применения древовидной структуры
<code>WeakHashMap</code>	Расширяет класс <code>AbstractMap</code> для применения хеш-таблицы со слабыми ключами
<code>LinkedHashMap</code>	Расширяет класс <code>HashMap</code> , разрешая итерацию с вводом элементов в определенном порядке
<code>IdentityHashMap</code>	Расширяет класс <code>AbstractMap</code> и использует результаты проверки ссылок на равенство при сравнении документов

Класс `HashMap`

Этот класс расширяет класс `AbstractMap` и реализует интерфейс `Map`. В нем используется хеш-таблица для хранения отображения, и благодаря этому обеспечивается постоянное время выполнения методов `get()` и `put()` даже в обращении к крупным отображениям. Класс `HashMap` является обобщенным и объявляется приведенным ниже образом, где `K` обозначает тип ключей, а `V` – тип хранимых в отображении значений.

```
class HashMap<K, V>
```

В классе определены следующие конструкторы:

```
HashMap()
HashMap(Map<? extends K, ? extends V> m)
HashMap(int емкость)
HashMap(int емкость, float коэффициент_заполнения)
```

В первой форме конструктора создается хеш-отображение по умолчанию. Во второй форме конструктора хеш-отображение инициируется элементами заданного отображения `m`. В третьей форме задается емкость хеш-отображения. И в четвертой форме емкость и коэффициент_заполнения хеш-отображения задаются в качестве аргументов конструктора. Назначение емкости и коэффициен-

610 Часть II. Библиотека Java

та заполнения такое же, как и в описанном ранее классе HashSet. По умолчанию емкость составляет 16, а коэффициент заполнения – 0,75.

Класс HashMap реализует интерфейс Map и расширяет класс AbstractMap, не дополняя их своими методами. Следует иметь в виду, что хеш-отображение не гарантирует порядок расположения своих элементов. Следовательно, порядок, в котором элементы вводятся в хеш-отображение, не обязательно соответствует тому порядку, в котором они извлекаются итератором. В следующем примере программы демонстрируется применение класса HashMap. В этой программе имена вкладчиков отображаются на остатки на их банковских счетах. Обратите внимание, каким образом получается и используется представление хеш-отображения в виде множества:

```
import java.util.*;  
  
class HashMapDemo {  
    public static void main(String args[]) {  
        // создать хеш-отображение  
        HashMap<String, Double> hm = new HashMap<String, Double>();  
  
        // ввести элементы в хеш-отображение  
        hm.put("Джон Доу", new Double(3434.34));  
        hm.put("Том Смит", new Double(123.22));  
        hm.put("Джейн Бейкер", new Double(1378.00));  
        hm.put("Тод Холл", new Double(99.22));  
        hm.put("Ральф Смит", new Double(-19.08));  
  
        // получить множество записей  
        Set<Map.Entry<String, Double>> set = hm.entrySet();  
  
        // вывести множество записей  
        for(Map.Entry<String, Double> me : set) {  
            System.out.print(me.getKey() + ": ");  
            System.out.println(me.getValue());  
        }  
        System.out.println();  
  
        // внести сумму 1000 на счет Джона Доу  
        double balance = hm.get("Джон Доу");  
        hm.put("Джон Доу", balance + 1000);  
        System.out.println("Новый остаток на счете Джона Доу: " +  
                           hm.get("Джон Доу"));  
    }  
}
```

Ниже приведен результат, выводимый данной программой (точный порядок следования записей может отличаться).

```
Ральф Смит: -19.08  
Том Смит: 123.22  
Джон Доу: 3434.34  
Тод Холл: 99.22  
Джейн Бейкер: 1378.0  
Новый остаток на счете Джона Доу: 4434.34
```

Выполнение данной программы начинается с создания хеш-отображения, в которое затем вводятся имена и фамилии вкладчиков, отображаемые на остатки на их банковских счетах. Далее содержимое хеш-отображения выводится

с помощью его представления в виде множества, получаемого из метода `entrySet()`. Ключи и значения выводятся в результате вызова методов `getKey()` и `getValue()`, определенных в интерфейсе `Map.Entry`. Обратите особое внимание на порядок внесения суммы на счет Джона Доу. Метод `put()` автоматически заменяет новым значением любое существовавшее ранее значение, связанное с указанным ключом. Таким образом, после обновления остатка на счете Джона Доу хеш-отображение по-прежнему содержит только один счет Джона Доу.

Класс `TreeMap`

Класс `TreeMap` расширяет класс `AbstractMap` и реализует интерфейс `NavigableMap`. В нем создается отображение, размещаемое в древовидной структуре. В классе `TreeMap` предоставляются эффективные средства для хранения пар “ключ-значение” в отсортированном порядке и обеспечивается их быстрое извлечение. Следует заметить, что, в отличие от хеш-отображения, древовидное отображение гарантирует, что его элементы будут отсортированы по порядку нарастания ключей. Класс `TreeMap` является обобщенным и объявляется следующим образом:

```
class TreeMap<K, V>
```

где `K` обозначает тип ключей, а `V` – тип хранимых в отображении значений. В классе `TreeMap` определены следующие конструкторы:

```
TreeMap()
TreeMap(Comparator<? super K> компаратор)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

В первой форме конструктора создается пустое древовидное отображение, которое будет отсортировано с естественным упорядочением ключей. Во второй форме конструктора создается пустое древовидное отображение, которое будет отсортировано с помощью заданного компаратора типа `Comparator`. (Компараторы обсуждаются далее в этой главе.) В третьей форме древовидное отображение инициализируется элементами из отображения `m`, которые будут отсортированы с естественным упорядочением ключей. И наконец, в четвертой форме создается древовидное отображение с элементами из отображения `sm`, которые будут отсортированы в том же порядке, что и в отображении `sm`.

В классе `TreeMap` не определяются дополнительные методы, помимо тех, что имеются в интерфейсе `NavigableMap` и классе `AbstractMap` для обращения с отображениями. Ниже приведен вариант предыдущего примера программы, переделанный с целью продемонстрировать применение класса `TreeMap`.

```
import java.util.*;

class TreeMapDemo {
    public static void main(String args[]) {
        // создать древовидное отображение
        TreeMap<String, Double> tm = new TreeMap<String, Double>();

        // ввести элементы в древовидное отображение
        tm.put("Джон Доу", new Double(3434.34));
        tm.put("Том Смит", new Double(123.22));
```

612 Часть II. Библиотека Java

```
tm.put("Джейн Бейкер", new Double(1378.00));
tm.put("Тод Халл", new Double(99.22));
tm.put("Ральф Смит", new Double(-19.08));

// получить множество записей
Set<Map.Entry<String, Double>> set = tm.entrySet();

// вывести множество записей
for(Map.Entry<String, Double> me : set) {
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}
System.out.println();

// внести сумму 1000 на счет Джона Доу
double balance = tm.get("Джон Доу");
tm.put("Джон Доу", balance + 1000);
System.out.println("Новый остаток на счете Джона Доу: " +
tm.get("Джон Доу"));
}
```

Ниже приведен результат, выводимый данной программой.

```
Джейн Бейкер: 1378.0
Джон Доу: 3434.34
Ральф Смит: -19.08
Тод Халл: 99.22
Том Смит: 123.22

Новый остаток на счете Джона Доу: 4434.34
```

Обратите внимание на то, что класс TreeMap сортирует ключи. Но в данном случае они сортируются по имени вместо фамилии. Такое поведение можно изменить, указав компаратор при создании отображения, как поясняется далее.

Класс LinkedHashMap

Класс LinkedHashMap расширяет класс HashMap. Он создает связный список элементов, расположенных в отображении в том порядке, в котором они вводились в него. Это позволяет организовать итерацию с вводом элементов в отображение в определенном порядке. Следовательно, при итерации представления отображения типа LinkedHashMap в виде коллекции его элементы будут возвращаться в том порядке, в котором они вводились в него. Можно также создать отображение типа LinkedHashMap, возвращающее свои элементы в том порядке, в котором к ним осуществлялся доступ в последний раз. Класс LinkedHashMap является обобщенным и объявляется приведенным ниже образом, где K обозначает тип ключей, а V – тип хранимых в отображении значений.

```
class LinkedHashMap<K, V>
```

В классе LinkedHashMap определяются следующие конструкторы:

```
LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int емкость)
LinkedHashMap(int емкость, float коэффициент_заполнения)
LinkedHashMap(int емкость, float коэффициент_заполнения, boolean порядок)
```

В первой форме конструктора создается отображение типа `LinkedHashMap` по умолчанию. Во второй форме конструктора отображение типа `LinkedHashMap` инициализируется элементами из заданного отображения `m`. В третьей форме задается емкость отображения, в четвертой – емкость и коэффициент_заполнения отображения. Назначение этих параметров такое же, как и у класса `HashMap`. По умолчанию емкость составляет 16, а коэффициент заполнения – 0,75. В последней форме конструктора можно указать порядок расположения элементов в связном списке: ввода или последнего доступа. Если параметр порядок принимает логическое значение `true`, то используется порядок доступа, а если он принимает логическое значение `false` – порядок ввода элементов.

В классе `LinkedHashMap` добавляется только один новый метод к тем, что определены в классе `HashMap`. Это метод `removeEldestEntry()`, общая форма которого приведена ниже.

```
protected boolean removeEldestEntry(Map.Entry<K, V> e)
```

Этот метод вызывается из методов `put()` и `putAll()`. Самая старая запись в отображении передается в качестве параметра `e`. По умолчанию этот метод возвращает логическое значение `false` и ничего не делает. Но если переопределить его, то можно удалить самую старую запись из отображения типа `LinkedHashMap`. Для этого переопределенный метод должен возвратить логическое значение `true`. А для того чтобы сохранить самую старую запись в отображении, из переопределенного метода следует возвратить логическое значение `false`.

Класс `IdentityHashMap`

Класс `IdentityHashMap` расширяет класс `AbstractMap` и реализует интерфейс `Map`. Он аналогичен классу `HashMap`, за исключением того, что при сравнении элементов отображения в нем выполняется проверка ссылок на равенство. Класс `IdentityHashMap` является обобщенным и объявляется следующим образом:

```
class IdentityHashMap<K, V>
```

где `K` обозначает тип ключей, а `V` – тип хранимых в отображении значений. В документации на прикладной программный интерфейс API ясно сказано, что класс `IdentityHashMap` не предназначен для общего применения.

Класс `EnumMap`

Класс `EnumMap` расширяет класс `AbstractMap` и реализует интерфейс `Map`. Он специально предназначен для применения вместе с ключами типа `enum`. Это обобщенный класс, объявляемый следующим образом:

```
class EnumMap<K extends Enum<K>, V>
```

где `K` обозначает тип ключей, а `V` – тип хранимых в отображении значений. Следует иметь в виду, что класс `K` должен расширять класс `Enum<K>`, а для этого ключи должны быть типа `enum`.

В классе `EnumMap` определены следующие конструкторы:

614 Часть II. Библиотека Java

```
EnumMap(Class<K> тип_ключа)
EnumMap(Map<K, ? extends V> m)
EnumMap(EnumMap<K, ? extends V> em)
```

Первый конструктор создает пустое отображение типа EnumMap для хранения элементов, имеющих заданный тип_ключа. Второй конструктор создает отображение типа EnumMap с теми же записями, что и в заданном отображении *m*. А третий конструктор создает отображение типа EnumMap, инициируемое значениями из отображения *em*. Свои методы в классе EnumMap не определяются.

Компараторы

Классы TreeSet и TreeMap сохраняют элементы в отсортированном порядке. Однако понятие “порядок сортировки” точно определяет применяемый ими компаратор. По умолчанию эти классы сохраняют элементы, используя то, что в Java называется *естественным упорядочением*, т.е. ожидаемым упорядочением, когда после **A** следует **B**, а после **1 – 2** и т.д. Если же элементы требуется упорядочить иным образом, то при создании множества или отображения следует указать компаратор типа Comparator. Это дает возможно точно управлять порядком сохранения элементов в отсортированных коллекциях.

Интерфейс Comparator является обобщенным и объявляется приведенным ниже образом, где **T** обозначает тип сравниваемых объектов.

```
interface Comparator<T>
```

До версии JDK 8 в интерфейсе Comparator определялись только два метода: compare() и equals(). Метод compare(), общая форма которого приведена ниже, сравнивает два элемента по порядку.

```
int compare(T объект1, T объект2)
```

Здесь параметры *объект1* и *объект2* обозначают сравниваемые объекты. Обычно этот метод возвращает нулевое значение, если объекты равны; положительное значение, если *объект1* больше, чем *объект2*, а иначе – отрицательное значение. Этот метод может генерировать исключение типа ClassCastException, если типы сравниваемых объектов несовместимы. Реализуя метод compare(), можно изменить порядок расположения объектов. Например, чтобы отсортировать объекты в обратном порядке, можно создать компаратор, который обращает результат их сравнения.

Метод equals(), общая форма которого приведена ниже, проверяет объект на равенство вызывающему компаратору.

```
boolean equals(object объект)
```

Здесь параметр *объект* обозначает проверяемый на равенство объект. Метод equals() возвращает логическое значение true, если заданный объект и вызывающий объект относятся к типу Comparator и упорядочиваются одним и тем же способом. В противном случае этот метод возвращает логическое значение false. Переопределение метода equals() не требуется, и большинство простых компараторов в этом не нуждается.

Многие годы в интерфейсе `Comparator` были доступны только оба упомянутых выше метода. Но после выпуска версии JDK 8 это положение коренным образом изменилось в лучшую сторону. В версии JDK 8 функциональные возможности интерфейса `Comparator` были значительно расширены благодаря внедрению методов по умолчанию и статических методов, рассматриваемых ниже по отдельности.

Используя метод `reversed()`, можно получить компаратор, изменяющий на обратное упорядочение сравниваемых объектов, с которым этот компаратор вызывался. Ниже приведена общая форма данного метода.

```
default Comparator<T> reversed()
```

Этот метод возвращает компаратор с обратным упорядочением. Так, если в исходном компараторе используется естественное упорядочение символов от **A** до **Z**, то компаратор с обратным упорядочением расположит букву **B** перед **A**, букву **C** перед **B** и т.д. С методом `reversed()` тесно связан метод `reverseOrder()`, общая форма которого выглядит следующим образом:

```
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

Этот метод возвращает компаратор, изменяющий на обратное естественное упорядочение сравниваемых элементов. С другой стороны, можно получить компаратор с естественным упорядочением сравниваемых элементов, вызвав статический метод `naturalOrder()`. Ниже приведена его общая форма.

```
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
```

Если же требуется компаратор, способный обрабатывать пустые значения `null`, то для этой цели служит метод `nullsFirst()` или `nullsLast()`. Ниже приведены общие формы этих методов.

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> компаратор)
static <T> Comparator<T> nullsLast(Comparator<? super T> компаратор)
```

Метод `nullsFirst()` возвращает компаратор, рассматривающий пустые значения `null` как меньшие остальных значений. А метод `nullsLast()` возвращает компаратор, рассматривающий пустые значения `null` как большие остальных значений. Но в любом случае заданный компаратор выполняет сравнение, если оба сравниваемых значения не являются пустыми. Если же заданному компаратору передается пустое значение `null`, то все непустые значения рассматриваются им как равнозначные.

В версии JDK 8 в интерфейсе `Comparator` появился еще один метод по умолчанию, выполняющий второе сравнение, если результат первого сравнения указывает на равенство сравниваемых объектов. Следовательно, с помощью этого метода можно составить последовательность “сравнить сначала по X, а затем по Y”. Например, при сравнении городов можно сначала сравнивать их названия, а затем названия штатов. Так, в естественном алфавитном порядке название Спрингфилд, Иллинойс, будет предшествовать названию Спрингфилд, Миссури. У метода `thenComparing()` имеются три общие формы объявления. Ниже приведена первая общая форма, позволяющая указать второй компаратор, передав экземпляр объекта типа `Comparator`. В этой форме параметр `второй_компаратор` обозначает компаратор, вызываемый в том случае, если первый компаратор возвращает признак равенства сравниваемых объектов.

```
default Comparator<T> thenComparing(Comparator<? super T> второй_компаратор)
```

616 Часть II. Библиотека Java

В двух других формах метода `thenComparing()` можно указать стандартный функциональный интерфейс `Function`, определенный в пакете `java.util.function`. Обе эти формы приведены ниже.

```
default <U extends Comparable<? super U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> получить_ключ)
default <U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> получить_ключ,
        Comparator<? super U> компаратор_ключей)
```

В обеих формах параметр `получить_ключ` обозначает функцию, получающую следующий ключ для сравнения. Этот ключ используется в том случае, если в результате первого сравнения возвращается признак равенства сравниваемых объектов. В последней форме данного метода параметр `компаратор_ключей` обозначает компаратор, используемый для сравнения ключей. (Здесь и далее `U` обозначает тип ключа.)

Интерфейс `Comparator` дополнен также специальными вариантами методов последующего сравнения примитивных типов данных. Их общие формы приведены ниже, где параметр `получить_ключ` обозначает функцию, получающую следующий ключ для сравнения.

```
default Comparator<T>
    thenComparingDouble(ToDoubleFunction<? super T> получить_ключ)
default Comparator<T>
    thenComparingIntToIntFunction<? super T> получить_ключ)
default Comparator<T>
    thenComparingLong(ToLongFunction<? super T> получить_ключ)
```

И наконец, в версии JDK 8 интерфейс `Comparator` дополнен методом `comparing()`. Этот метод возвращает компаратор, получающий ключ для сравнения от функции, передаваемой данному методу в качестве параметра. Ниже приведены обе формы объявления метода `comparing()`.

```
static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> получить_ключ)
static <T, U> Comparator<T>
    comparing(Function<? super T, ? extends U> получить_ключ,
        Comparator<? super U> компаратор_ключей)
```

В обеих формах параметр `получить_ключ` обозначает функцию, получающую следующий ключ для сравнения. Во второй форме данного метода параметр `компаратор_ключей` обозначает компаратор, используемый для сравнения ключей. Интерфейс `Comparator` дополнен также специальными вариантами метода `comparing()` для сравнения примитивных типов данных. Их общие формы приведены ниже, где параметр `получить_ключ` обозначает функцию, получающую следующий ключ для сравнения.

```
static <T> Comparator<T>
    ComparingDouble(ToDoubleFunction<? super T> получить_ключ)
static <T> Comparator<T>
    ComparingIntToIntFunction<? super T> получить_ключ)
static <T> Comparator<T>
    ComparingLong(ToLongFunction<? super T> получить_ключ)
```

Применение компараторов

Ниже приведен пример программы, демонстрирующий эффективность специальных компараторов. В этой программе реализуется метод `compare()` для сравнения символьных строк в порядке, обратном обычному. Это означает, что элементы древовидного множества сортируются в обратном порядке.

```
// Использовать специальный компаратор
import java.util.*;

// Компаратор для сравнения символьных строк в обратном порядке
class MyComp implements Comparator<String> {
    public int compare(String a, String b) {
        String aStr, bStr;
        aStr = a;
        bStr = b;
        // выполнить сравнение в обратном порядке
        return bStr.compareTo(aStr);
    }
    // переопределять метод equals() не требуется
}

class CompDemo {
    public static void main(String args[]) {
        // создать древовидное множество типа TreeSet
        TreeSet<String> ts = new TreeSet<String>(new MyComp());
        // ввести элементы в древовидное множество
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");

        // вывести элементы из древовидного множества
        for(String element : ts)
            System.out.print(element + " ");
        System.out.println();
    }
}
```

Как показывает приведенный ниже результат выполнения данной программы, древовидное множество теперь отсортировано в обратном порядке.

F E D C B A

Обратите внимание на класс `MyComp`, реализующий интерфейс `Comparator` и метод `compare()`. Как упоминалось выше, переопределение метода `equals()` не требуется и вообще не принято. Переопределение не требуется и методам по умолчанию, внедренным в интерфейсы, начиная с версии JDK 8.) В теле метода `compare()` для объекта типа `String` вызывается метод `compareTo()`, сравнивающий две символьные строки. Но метод `compareTo()` вызывается для строкового объекта `bStr`, а не `aStr`. Благодаря этому результат сравнения получается обратным.

Несмотря на то что способ реализации компаратора с обратным упорядочением в предыдущем примере программы вполне пригоден, в версии JDK 8 появился еще один способ сделать то же самое – просто вызвать метод `reversed()` для ком-

618 Часть II. Библиотека Java

паратора с естественным упорядочением. Этот метод возвратит эквивалентный компаратор, который, однако, сравнивает объекты в обратном порядке. Так, предыдущий пример программы можно переделать, внеся следующие изменения в класс MyClass:

```
class MyComp implements Comparator<String> {
    public int compare(String aStr, String bStr)
        return aStr.compareTo(bStr);
}
```

Далее можно воспользоваться следующим фрагментом кода для создания древовидного множества типа TreeSet, где строчные элементы располагаются в обратном порядке:

```
MyComp mc = new MyComp(); // создать компаратор
// передать вариант компаратора типа MyComp с обратным
// упорядочением древовидному множеству типа TreeSet
TreeSet<String> ts = new TreeSet<String>(mc.reversed());
```

Если ввести этот новый код в предыдущий пример программы, то будет получен тот же самый результат ее выполнения. В данном примере применение метода reversed() не дает никаких преимуществ, но в тех случаях, когда требуется создать компараторы с естественным и обратным упорядочением, метод reversed() упрощает получение компаратора с обратным упорядочением, не требуя написания дополнительного кода специально для этой цели.

Начиная с версии JDK 8 создавать класс MyComp, как показано в предыдущих примерах, фактически не требуется, поскольку его можно легко заменить соответствующим лямбда-выражением. В частности, класс MyComp можно полностью удалить, а вместо него создать компаратор символьных строк, используя следующий фрагмент кода:

```
// использовать лямбда-выражение для реализации
// компаратора типа Comparator<String>
Comparator<String> mc = (aStr, bStr) -> aStr.compareTo(bStr);
```

Следует также заметить, что в данном простом примере компаратор с обратным упорядочением можно задать и с помощью лямбда-выражения непосредственно в вызове конструктора класса TreeSet, как показано ниже.

```
// передать компаратор с обратным упорядочением конструктору
// класса TreeSet через лямбда-выражение
TreeSet<String> ts = new TreeSet<String>(
    (aStr, bStr) -> bStr.compareTo(aStr));
```

Внеся эти изменения, можно значительно сократить исходный код упомянутой выше программы, как показывает приведенный ниже окончательный ее вариант.

```
// Использовать лямбда-выражение для создания компаратора
// с обратным упорядочением
import java.util.*;

class CompDemo2 {
    public static void main(String args[]) {
        // передать компаратор с обратным упорядочением
```

```

// древовидному множеству типа TreeSet
TreeSet<String> ts = new TreeSet<String>(
    (aStr, bStr) -> bStr.compareTo(aStr));

// ввести элементы в древовидное множество
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
// вывести элементы из древовидного множества
for(String element : ts)
    System.out.print(element + " ");

System.out.println();
}
}

```

Рассмотрим более полезное применение компараторов на примере переделанного варианта представленной ранее программы, где имена и фамилии вкладчиков и остатки на их банковских счетах сохраняются в древовидном отображении типа TreeMap. В предыдущем варианте этой программы счета сортировались по имени каждого вкладчика, а в новом, приведенном ниже ее варианте – по его фамилии. Для этого в ней используется компаратор, сравнивающий фамилии каждого вкладчика. В итоге получается отображение, отсортированное по фамилиям вкладчиков.

```

// Использовать компаратор для сортировки счетов
// по фамилиям вкладчиков
import java.util.*;

// сравнить последние слова в обеих символьных строках
class TComp implements Comparator<String> {
    public int compare(String aStr, String bStr) {
        int i, j, k;

        // найти индекс символа, с которого начинается фамилия
        i = aStr.lastIndexOf(' ');
        j = bStr.lastIndexOf(' ');
        k = aStr.substring(i).compareTo(bStr.substring(j));
        if(k==0) // Фамилии совпадают, проверить имя и фамилию полностью
            return aStr.compareTo(bStr);
        else
            return k;
    }
    // переопределять метод equals() не требуется
}

class TreeMapDemo2 {
    public static void main(String args[]) {
        // создать древовидное отображение
        TreeMap<String, Double> tm =
            new TreeMap<String, Double>(new TComp());

        // ввести элементы в древовидное отображение
        tm.put("Джон Доу", new Double(3434.34));
        tm.put("Том Смит", new Double(123.22));
        tm.put("Джейн Бейкер", new Double(1378.00));
    }
}

```

620 Часть II. Библиотека Java

```
tm.put("Тод Халл", new Double(99.22));
tm.put("Ральф Смит", new Double(-19.08));

// получить множество элементов
Set<Map.Entry<String, Double>> set = tm.entrySet();

// вывести элементы из множества
for(Map.Entry<String, Double> me : set) {
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}

System.out.println();

// внести сумму 1000 на счет Джона Доу
double balance = tm.get("Джон Доу");
tm.put("Джон Доу ", balance + 1000);
System.out.println("Новый остаток на счете Джона Доу: " +
    tm.get("Джон Доу"));
}
}
```

Ниже приведен результат, выводимый данной программой. Обратите внимание на то, что счета теперь отсортированы по фамилиям вкладчиков.

```
Джейн Бейкер: 1378.0
Джон Доу: 3434.34
Ральф Смит: -19.08
Том Смит: 123.22
Тод Халл: 99.22

Новый остаток на счете Джона Доу: 4434.34
```

Компаратор типа `TComp` сравнивает две символьные строки, содержащие имя и фамилию. Сначала он сравнивает фамилии. Для этого осуществляется поиск позиции последнего пробела в каждой строке с последующим сравнением подстроки, начинающейся с этой позиции. В том случае, если фамилии одинаковы, сравниваются имена. В итоге получается древовидное отображение, отсортированное по фамилиям вкладчиков, а в пределах одинаковых фамилий – по именам. Можете убедиться в этом сами, поскольку имя и фамилия Ральф Смит в приведенном выше результате выполнения данной программы располагаются выше имени и фамилии Том Смит.

Начиная с версии JDK 8 предыдущий пример программы можно переделать таким образом, чтобы отсортировать древовидное отображение сначала по фамилии, а затем по имени вкладчика, используя метод `thenComparing()`. Напомним, что метод `thenComparing()` позволяет указать второй компаратор, который используется в том случае, если вызывающий компаратор возвращает признак равенства сравниваемых объектов. Именно такой способ сортировки счетов вкладчиков реализован в приведенном ниже переделанном варианте предыдущего примера программы.

```
// Использовать метод thenComparing() для сортировки
// счетов вкладчиков сначала по фамилии, а затем по имени
import java.util.*;
```

```
// Компаратор, сравнивающий фамилии вкладчиков
```

```

class CompLastNames implements Comparator<String> {
    public int compare(String aStr, String bStr) {
        int i, j;

        // найти индекс символа, с которого начинается фамилия
        i = aStr.lastIndexOf(' ');
        j = bStr.lastIndexOf(' ');
        return aStr.substring(i).compareToIgnoreCase(bStr.substring(j));
    }
}

// отсортировать счета вкладчиков по Ф.И.О., если фамилии одинаковы
class CompThenByFirstName implements Comparator<String> {
    public int compare(String aStr, String bStr) {
        int i, j;

        return aStr.compareToIgnoreCase(bStr);
    }
}

class TreeMapDemo2A {
    public static void main(String args[]) {
        // использовать метод thenComparing() для создания
        // компаратора, сравнивающего сначала фамилии, а затем
        // Ф.И.О. вкладчиков, если фамилии одинаковы
        CompLastNames compLN = new CompLastNames();
        Comparator<String> compLastThenFirst =
            compLN.thenComparing(new CompThenByFirstName());

        // создать древовидное отображение
        TreeMap<String, Double> tm =
            new TreeMap<String, Double>(compLastThenFirst);
        // ввести элементы в древовидное отображение
        tm.put("Джон Доу", new Double(3434.34));
        tm.put("Том Смит", new Double(123.22));
        tm.put("Джейн Бейкер", new Double(1378.00));
        tm.put("Тод Халл", new Double(99.22));
        tm.put("Ральф Смит", new Double(-19.08));

        // получить множество элементов
        Set<Map.Entry<String, Double>> set = tm.entrySet();

        // вывести элементы из множества
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        // внести сумму 1000 на счет Джона Доу
        double balance = tm.get("Джон Доу");
        tm.put("Джон Доу", balance + 1000);

        System.out.println("Новый остаток на счете Джона Доу: " +
            tm.get("Джон Доу"));
    }
}

```

Результат выполнения данного варианта рассматриваемой здесь программы такой же, как и предыдущего ее варианта. Их отличие лишь в том, как реализуется сортировка счетов вкладчиков. Прежде всего обратите внимание на создание компаратора

типа CompLastNames. Этот компаратор сравнивает только фамилии вкладчиков, тогда как второй компаратор типа CompThenByFirstName – Ф.И.О. вкладчиков, начиная с имени. Ниже показано, каким образом создаются оба эти компаратора.

```
CompLastNames compLN = new CompLastNames();
Comparator<String> compLastThenFirst =
    compLN.thenComparing(new CompThenByFirstName());
```

Здесь первый компаратор присваивается переменной compLN в виде экземпляра класса CompLastNames. Для него вызывается метод thenComparing(), которому в качестве параметра передается экземпляр класса CompThenByFirstName. Полученный результат присваивается переменной compLastThenFirst. Этот второй компаратор используется для построения древовидного множества типа TreeMap, как показано ниже.

```
TreeMap<String, Double> tm =
    new TreeMap<String, Double>(compLastThenFirst);
```

Теперь сортировка счетов вкладчиков выполняется по Ф.И.О., если их фамилии оказываются одинаковыми. Это означает, что имена вкладчиков упорядочиваются по фамилиям, а в пределах одинаковых фамилий – по именам.

И последнее замечание: ради наглядности в данном примере оба компаратора создаются явным образом в виде классов CompLastNames и ThenByFirstNames, но вместо них можно воспользоваться лямбда-выражениями. Попробуйте сделать это сами в качестве упражнения, следуя тому же самому общему образцу из приведенного выше примера с классом CompDemo2.

Алгоритмы коллекций

В каркасе коллекций определяется ряд алгоритмов, которые можно применять к коллекциям и отображениям. Эти алгоритмы определены в виде статических методов из класса Collections, перечисленных в табл. 18.19. Как упоминалось ранее, в версии JDK 5 все алгоритмы были переделаны с учетом обобщений.

Таблица 18.19. Алгоритмы, определенные в классе Collections

Метод	Описание
<code>static <T> boolean addAll(Collection<? super T> c, T... элементы)</code>	Вставляет заданные <i>элементы</i> в указанную коллекцию <i>c</i> . Возвращает логическое значение <code>true</code> , если элементы были добавлены, а иначе – логическое значение <code>false</code>
<code>static <T> Queue<T> asLifoQueue(Deque<T> c)</code>	Возвращает представление заданной коллекции <i>c</i> в виде стека, действующего по принципу “последним пришел – первым обслужен”
<code>static <T> int binarySearch(List<? extends T> список, T значение, Comparator<? super T> c)</code>	Осуществляет поиск указанного <i>значения</i> в заданном <i>списке</i> , упорядоченном в соответствии заданным компаратором <i>c</i> . Возвращает позицию указанного <i>значения</i> в заданном <i>списке</i> или отрицательное значение, если <i>значение</i> не найдено

Продолжение табл. 18.19

Метод	Описание
<code>static <T> int binarySearch(List<? Extends Comparable<? super T>> список, T значение)</code>	Осуществляет поиск указанного значения в заданном списке , который должен быть отсортирован. Возвращает позицию указанного значения в заданном списке или отрицательное значение, если значение не найдено
<code>static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> t)</code>	Возвращает динамически типизируемое представление коллекции. Попытка ввести несовместимый элемент в коллекцию вызовет исключение типа ClassCastException
<code>static <E> List<E> checkedList(List<E> c, Class<E> t)</code>	Возвращает динамически типизируемое представление списка типа List . Попытка ввести несовместимый элемент в список вызовет исключение типа ClassCastException
<code>static <K, V> Map<K, V>checkedMap(Map<K, V> c, Class<E> тип_ключа, Class<V> тип_значения)</code>	Возвращает динамически типизируемое представление отображения типа Map . Попытка ввести несовместимый элемент в отображение вызовет исключение типа ClassCastException
<code>static <K, V> NavigableMap<K, V> checkedNavigableMap (NavigableMap<K, V> nm, Class<E> тип_ключа, Class<V> тип_значения)</code>	Возвращает динамически типизируемое представление отображения типа NavigableMap . Попытка ввести несовместимый элемент в отображение вызовет исключение типа ClassCastException (добавлен в версии JDK 8)
<code>static <E> NavigableSet<E> checkedNavigableSet (NavigableSet<E> ns, Class<E> t)</code>	Возвращает динамически типизируемое представление в множество типа NavigableSet . Попытка ввести несовместимый элемент в множество вызовет исключение типа ClassCastException (добавлен в версии JDK 8)
<code>static <E> Queue<E> checkedQueue(Queue<E> q, Class<E> t)</code>	Возвращает динамически типизируемое представление очереди типа checkedQueue . Попытка ввести несовместимый элемент в множество вызовет исключение типа ClassCastException (добавлен в версии JDK 8)
<code>static <E> List<E> checkedSet(Set<E> c, Class<E> t)</code>	Возвращает динамически типизируемое представление множества типа Set . Попытка ввести несовместимый элемент в множество вызовет исключение типа ClassCastException
<code>static <K, V> SortedMap<K, V>checkedSortedMap(SortedMap<K, V> c, Class<E> тип_ключа, Class<V> тип_значения)</code>	Возвращает динамически типизируемое представление отображения типа SortedMap . Попытка ввести несовместимый элемент в отображение вызовет исключение типа ClassCastException
<code>static <E> SortedSet<E> checkedSortedSet(SortedSet<E> c, Class<E> t)</code>	Возвращает динамически типизируемое представление множества типа SortedSet . Попытка ввести несовместимый элемент в множество вызовет исключение типа ClassCastException

Продолжение табл. 18.19

Метод	Описание
<code>static <T> void copy(List<? super T> список1, List<? Extends T> список2)</code>	Копирует элементы из <i>списка2</i> в <i>список1</i>
<code>static boolean disjoint(Collection<?> a, Collection<?> b)</code>	Сравнивает элементы коллекций <i>a</i> и <i>b</i> . Возвращает логическое значение <code>true</code> , если обе коллекции не содержат общие элементы (т.е. не пересекающиеся множества элементов), а иначе – логическое значение <code>false</code>
<code>static <T> Enumeration<T> emptyEnumeration()</code>	Возвращает пустое перечисление, т.е. перечисление без элементов
<code>static <T> Iterator<T> emptyIterator()</code>	Возвращает пустой итератор, т.е. итератор без элементов
<code>static <T> List<T> emptyList()</code>	Возвращает неизменяемый, пустой список типа, выводимого из интерфейса <code>List</code>
<code>static <T> ListIterator<T> emptyListIterator()</code>	Возвращает пустой итератор списка, т.е. итератор списка без элементов
<code>static <K, V> Map<K, V> emptyMap()</code>	Возвращает неизменяемое, пустое отображение типа, выводимого из интерфейса <code>Map</code>
<code>static <K, V> NavigableMap<K, V> emptyNavigableMap()</code>	Возвращает неизменяемое, пустое отображение типа, выводимого из интерфейса <code>NavigableMap</code> (добавлен в версии JDK 8)
<code>static <E> NavigableSet<E> emptyNavigableSet()</code>	Возвращает неизменяемое, пустое множество типа, выводимого из интерфейса <code>NavigableSet</code> (добавлен в версии JDK 8)
<code>static <T> Set<T> emptySet()</code>	Возвращает неизменяемое, пустое множество типа, выводимого из интерфейса <code>Set</code>
<code>static <K, V> SortedMap<K, V> emptySortedMap()</code>	Возвращает неизменяемое, пустое отображение типа, выводимого из интерфейса <code>SortedMap</code> (добавлен в версии JDK 8)
<code>static <E> SortedSet<E> emptySortedSet()</code>	Возвращает неизменяемое, пустое множество типа, выводимого из интерфейса <code>SortedSet</code> (добавлен в версии JDK 8)
<code>static <T> Enumeration<T> enumeration(Collection<T> c)</code>	Возвращает перечисление элементов из заданной коллекции <i>c</i> . (См. раздел “Интерфейс <code>Enumeration</code> ” далее в этой главе)
<code>static <T> void fill(List<? super T> список, Т объект)</code>	Присваивает указанный <i>объект</i> каждому элементу заданного <i>списка</i>
<code>static int frequency(Collection<?> c, object объект)</code>	Подсчитывает количество вхождений указанного <i>объекта</i> в заданной коллекции <i>c</i> и возвращает результат
<code>static int indexOfSubList(List<?> список, List<?> подсписок)</code>	Осуществляет поиск первого вхождения указанного <i>подсписка</i> в заданный <i>список</i> . Возвращает индекс первого совпадения или значение <code>-1</code> , если совпадение не обнаружено

Продолжение табл. 18.19

Метод	Описание
<code>static int lastIndexOfSubList(List<?> список, List<?> подсписок)</code>	Осуществляет поиск последнего вхождения указанного <i>подсписка</i> в заданный <i>список</i> . Возвращает индекс первого совпадения или значение <code>-1</code> , если совпадение не обнаружено
<code>static <T> ArrayList<T> list(Enumeration<T> перечисление)</code>	Возвращает списочный массив типа <code>ArrayList</code> , содержащий элементы заданного <i>перечисления</i>
<code>static <T> T max(Collection<? extends T> c, Comparator<? super T> компаратор)</code>	Возвращает максимальный элемент из указанной коллекции <i>c</i> , определяемый заданным <i>компаратором</i>
<code>static <T extends Comparable<? super T>> T max(Collection<? extends T> c)</code>	Возвращает максимальный элемент из указанной коллекции <i>c</i> , определяемый естественным упорядочением. Коллекция должна быть отсортированной
<code>static <T> T min(Collection<? extends T> c, Comparator<? super T> компаратор)</code>	Возвращает минимальный элемент из указанной коллекции <i>c</i> , определяемый заданным <i>компаратором</i> . Коллекция необязательно должна быть отсортированной
<code>static <T extends Comparable<? super T>> T min(Collection<? extends T> c)</code>	Возвращает минимальный элемент из указанной коллекции <i>c</i> , определяемый естественным упорядочением
<code>static <T> List<T> nCopies(int количество, T объект)</code>	Возвращает заданное <i>количество</i> копий указанного <i>объекта</i> , содержащихся в неизменяемом списке. Значение параметра <i>количество</i> должно быть больше или равно нулю
<code>static <E> Set<E> newSetFromMap(Map<E, Boolean> m)</code>	Создает и возвращает множество, исходя из указанного отображения <i>m</i> , которое должно быть пустым на момент вызова данного метода
<code>static <T> boolean replaceAll(List<T> список, T старый, T новый)</code>	Заменяет все вхождения заданного элемента <i>старый</i> на элемент <i>новый</i> в указанном <i>списке</i> . Возвращает логическое значение <code>true</code> , если произведена хотя бы одна замена, а иначе — логическое значение <code>false</code>
<code>static void reverse(List<T> список)</code>	Изменяет на обратный порядок следования элементов в указанном <i>списке</i>
<code>static <T> Comparator<T> reverseOrder(Comparator<T> компаратор)</code>	Возвращает компаратор, обратный переданному <i>компаратору</i> . Следовательно, возвращаемый компаратор обращает результат сравнения, выполненного заданным <i>компаратором</i>
<code>static <T> Comparator<T> reverseOrder()</code>	Возвращает обратный компаратор, который обращает результат сравнения двух элементов коллекции
<code>static void rotate(List<T> список, int n)</code>	Смещает указанный <i>список</i> на <i>n</i> позиций вправо. Для смещения влево следует указать отрицательное значение параметра <i>n</i>

Продолжение табл. 18.19

Метод	Описание
<code>static void shuffle(List<T> список, Random r)</code>	Перетасовывает (случайным образом) элементы указанного <i>списка</i> , используя значение параметра <i>r</i> в качестве исходного для получения случайных чисел
<code>static void shuffle(List<T> список)</code>	Перетасовывает (случайным образом) элементы указанного <i>списка</i>
<code>static <T> Set<T> singleton(T объект)</code>	Возвращает заданный <i>объект</i> в виде неизменяемого множества. Это простейший способ преобразовать одиничный объект в множество
<code>static <T> List<T> singletonList(T объект)</code>	Возвращает заданный <i>объект</i> в виде неизменяемого списка. Это простейший способ преобразовать одиничный объект в список
<code>static <K, V> Map<K, V> singletonMap(K k, V v)</code>	Возвращает пару <i>k, v</i> “ключ–значение” в виде неизменяемого отображения. Это простейший способ преобразовать пару “ключ–значение” в отображение
<code>static <T> void sort(List<T> список, Comparator<? super T> компаратор)</code>	Сортирует элементы указанного <i>списка</i> в соответствии с заданным <i>компаратором</i>
<code>static <T extends Comparable<? super T>> void sort(List<T> список)</code>	Сортирует элементы указанного <i>списка</i> в соответствии с естественным упорядочением
<code>static void swap(List<T> список, int индекс1, int индекс2)</code>	Меняет местами элементы указанного <i>списка</i> , обозначаемые параметрами <i>индекс1</i> и <i>индекс2</i>
<code>static <T> Collection<T> synchronizedCollection(Collection<T> c)</code>	Возвращает потокобезопасную коллекцию, исходя из указанной коллекции <i>c</i>
<code>static <T> List<T> synchronizedList(List<T> список)</code>	Возвращает потокобезопасный список, исходя из указанного <i>списка</i>
<code>static <K, V> Map<K, V> synchronizedMap(Map<K, V> m)</code>	Возвращает потокобезопасное отображение, исходя из указанного отображения <i>m</i>
<code>static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> nm)</code>	Возвращает синхронизированное навигационное отображение, исходя из указанного отображения <i>nm</i> (добавлен в версии JDK 8)
<code>static <T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> ns)</code>	Возвращает синхронизированное навигационное множество, исходя из указанного множества <i>ns</i> (добавлен в версии JDK 8)
<code>static <T> Set<T> synchronizedSet(Set<T> s)</code>	Возвращает потокобезопасное множество, исходя из указанного множества <i>s</i>
<code>static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> sm)</code>	Возвращает потокобезопасное отсортированное отображение, исходя из указанного отображения <i>sm</i>

Окончание табл. 18.19

Метод	Описание
<code>static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> ss)</code>	Возвращает потокобезопасное отсортированное множество, исходя из указанного множества <code>ss</code>
<code>static <T> Collection<T> unmodifiable Collection (Collection<? extends T> c)</code>	Возвращает неизменяемую коллекцию, исходя из указанной коллекции <code>c</code>
<code>static <T> List<T> unmodifiableList(List<? extends T> список)</code>	Возвращает неизменяемый список, исходя из указанного списка
<code>static <K, V> Map<K, V>unmodifiableMap(Map<? extends K, ? extends V> m)</code>	Возвращает неизменяемое отображение, исходя из указанного отображения <code>m</code>
<code>static <K, V> NavigableMap<K, V> unmodifiableNavigableMap (NavigableMap<K, ? extends V> nm)</code>	Возвращает неизменяемое навигационное отображение, исходя из указанного отображения <code>nm</code> (добавлен в версии JDK 8)
<code>static <T> NavigableSet<T> unmodifiableNavigableSet (NavigableSet<T> ns)</code>	Возвращает неизменяемое навигационное множество, исходя из указанного множества <code>ns</code> (добавлен в версии JDK 8)
<code>static <T> Set<T> unmodifiableSet(Set<? extends T> s)</code>	Возвращает неизменяемое множество, исходя из указанного множества <code>s</code>
<code>static <K, V> SortedMap<K, V>unmodifiableSortedMap (SortedMap<K, ? extends V> sm)</code>	Возвращает неизменяемое отсортированное отображение, исходя из указанного отображения <code>sm</code>
<code>static <T> SortedSet<T> unmodifiableSortedSet (SortedSet<T> ss)</code>	Возвращает неизменяемое отсортированное множество, исходя из указанного множества <code>ss</code>

При попытке сравнить несовместимые типы некоторые из перечисленных выше методов могут сгенерировать исключение типа `ClassCastException`, а при попытке видоизменить неизменяемые коллекции — исключение типа `UnsupportedOperationException`. В зависимости от конкретного метода возможны и другие исключения.

Особое внимание следует уделить ряду *роверяемых* методов вроде метода `checkedCollection()`, возвращающего то, что в документации на прикладной программный интерфейс API называется “динамически типизируемым представлением” коллекций. Такое представление служит ссылкой на коллекцию, которая во время выполнения контролирует вводимые в коллекцию объекты на предмет совместимости типов. Любая попытка ввести в коллекцию несовместимый объект вызовет исключение типа `ClassCastException`. Пользоваться этим представлением удобно на стадии отладки, поскольку оно гарантирует наличие в коллекции достоверных элементов. К числу проверяемых относятся методы `checkedSet()`, `checkedList()`, `checkedMap()` и т.д. Они позволяют получить динамически типизированное представление указанной коллекции.

Следует заметить, что некоторые методы наподобие `synchronizedList()` и `synchronizedSet()` служат для получения синхронизированных (*потокобезопасных*) копий различных коллекций. Как упоминалось ранее, стандартные реализа-