```
Емкость вектора после ввода четырех элементов: 5
Текущая емкость вектора: 5
Текущая емкость вектора: 7
Текущая емкость вектора: 9
Первый элемент вектора: 1
Последний элемент вектора: 12
Вектор содержит 3.

Элементы вектора:
1 2 3 4 5 6 7 9 10 11 12
```

Вместо того чтобы полагаться на перебор объектов в цикле, как это делается в приведенном выше примере программы, можно воспользоваться итератором. Для этого в данную программу можно, например, подставить следующий фрагмент кода:

```
// Использовать итератор для вывода содержимого вектора
Iterator<Integer> vItr = v.iterator();

System.out.println("\nЭлементы вектора:");
while(vItr.hasNext())
    System.out.print(vItr.next() + " ");
System.out.println();
```

Для перебора элементов вектора можно также организовать цикл `for` в стиле `for each`, как показано в следующей версии предыдущего фрагмента кода:

```
// Использовать усовершенствованный цикл for в стиле for each
// для вывода элементов вектора
System.out.println("\nЭлементы вектора:");
for(int i : v)
    System.out.print(i + " ");

System.out.println();
```

Интерфейс `Enumeration` не рекомендуется применять в новом коде, поэтому для перебора всех элементов вектора обычно применяются итераторы и циклы `for` в стиле `for each`. Безусловно, существует еще немалый объем кода, в котором используется интерфейс `Enumeration`. Правда, перечисления и итераторы действуют практически одинаково.

Класс `Stack`

Класс `Stack` является производным от класса `Vector` и реализует стандартный стек, действующий по принципу "последним пришел — первый обслужен". В классе `Stack` определяется только конструктор по умолчанию, создающий пустой стек. В версии JDK 5 класс `Stack` был переделан под синтаксис обобщений и теперь объявляется следующим образом:

```
class Stack<E>
```

где **E** обозначает тип элементов, сохраняемых в стеке. Класс `Stack` включает в себя все методы, определенные в классе `Vector`, и дополняет их рядом своих методов, перечисленных в табл. 18.21.

Таблица 18.21. Методы из класса `Stack`

Метод	Описание
`boolean empty()`	Возвращает логическое значение **`true`**, если стек пустой, а если он содержит элементы, то логическое значение **`false`**
`E peek()`	Возвращает элемент из вершины стека, но не удаляет его
`E pop()`	Возвращает элемент из вершины стека, удаляя его
`E push(E` *элемент*`)`	Размещает заданный *элемент* в стеке и возвращает этот *элемент*
`int search(Object` *элемент*`)`	Осуществляет поиск заданного *элемента* в стеке. Если заданный *элемент* найден, возвращается его смещение относительно вершины стека, а иначе — значение **`-1`**

Чтобы разместить объект на вершине стека, следует вызвать метод `push()`; чтобы удалить и возвратить верхний элемент из стека — метод `pop()`, а для возврата верхнего элемента без его удаления из стека — метод `peek()`. Исключение типа `EmptyStackException` генерируется в том случае, если вызвать метод `pop()` или `peek()` для пустого стека. Метод `empty()` возвращает логическое значение `true`, если стек пуст. Метод `search()` определяет, содержится ли объект в стеке, и возвращает количество вызовов метода `pop()`, требующихся для перемещения объекта на вершину стека. Ниже приведен пример программы, в которой создается стек и в нем размещается несколько объектов типа `Integer`, а затем они извлекаются из стека.

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

class StackDemo {
    static void showpush(Stack<Integer> st, int a) {
        st.push(a);
        System.out.println("push(" + a + ")");
        System.out.println("стек: " + st);
    }

    static void showpop(Stack<Integer> st) {
        System.out.print("pop -> ");
        Integer a = st.pop();
        System.out.println(a);
        System.out.println("стек: " + st);
    }

    public static void main(String args[]) {
        Stack<Integer> st = new Stack<Integer>();

        System.out.println("стек: " + st);

        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop(st);
        showpop(st);

        try {
            showpop(st);
        } catch (EmptyStackException e) {
            System.out.println("стек пуст");
        }
    }
}
```

Окончание табл. 18.5

Метод	Описание
NavigableSet<E> tailSet(E *нижняя_граница*, **boolean** *включительно*)	Возвращает объект типа **NavigableSet**, включающий все элементы из вызывающего множества, которые больше заданной *нижней_границы*. Если же параметр *включительно* принимает логическое значение **true**, то в результирующее множество включается элемент, равный заданной *нижней_границе*. 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
E element()	Возвращает элемент из головы очереди. Возвращаемый элемент не удаляется. Если очередь пуста, генерируется исключение типа **NoSuchElementException**
boolean offer(E *объект*)	Пытается ввести заданный *объект* в очередь. Возвращает логическое значение **true**, если *объект* введен, а иначе – логическое значение **false**
E peek()	Возвращает элемент из головы очереди. Если очередь пуста, возвращает пустое значение **null**. Возвращаемый элемент не удаляется из очереди
E poll()	Возвращает элемент из головы очереди и удаляет его. Если очередь пуста, возвращает пустое значение **null**.
E remove()	Удаляет элемент из головы очереди, возвращая его. Генерирует исключение типа **NoSuchElementException**, если очередь пуста

Некоторые методы из данного интерфейса генерируют исключение типа `ClsssCastException`, 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 `IllegalArgimentException` 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 Deque<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 IllegalArgimentException. 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
void addFirst(E *объект***)**	Вводит заданный *объект* в голову двусторонней очереди. Генерирует исключение типа **IllegalStateException**, если в очереди фиксированной длины нет свободного места
void addLast(E *объект***)**	Вводит заданный *объект* в хвост двусторонней очереди. Генерирует исключение типа **IllegalStateException**, если в очереди фиксированной длины нет свободного места

Продолжение табл. 18.7

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

Окончание табл. 18.7

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

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.