Глава 10
КОЛЛЕКЦИИ

Общие определения

Коллекции – это хранилища, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных типов (структур) данных, поддерживающих три основные операции:

- добавление нового элемента в коллекцию;
- удаление элемента из коллекции;
- изменение элемента в коллекции.

В качестве других операций могут быть реализованы следующие: просмотреть элементы, подсчитать их количество и др.

Применение коллекций обусловливается возросшими объемами обрабатываемой информации. Когда счет используемых объектов идет на сотни тысяч, массивы не обеспечивают ни должной скорости, ни экономии ресурсов. Например, процессор UltraSPARC T1 тестировался на обработке информации для электронного магазина, содержащего около 40 тысяч товаров и 125 миллионов клиентов, сделавших 400 миллионов заказов.

Примером коллекции является стек (структура LIFO – Last In First Out), в котором всегда удаляется объект, вставленный последним. Для очереди (структура FIFO – First In First Out) используется другое правило удаления: всегда удаляется элемент, вставляемый первым. В абстрактных типах данных существует несколько видов очередей: двусторонние очереди, кольцевые очереди, обобщенные очереди, в которых запрещены повторяющиеся элементы. Стеки и очереди могут быть реализованы как на базе массива, так и на базе связного списка.

Коллекции в языке Java объединены в библиотеке классов **`java.util`** и представляют собой контейнеры для хранения и манипулирования объектами. До появления Java 2 эта библиотека содержала классы только для работы с простейшими структурами данных: **`Vector`**, **`Stack`**, **`Hashtable`**, **`BitSet`**, а также интерфейс **`Enumeration`** для работы с элементами этих классов. Коллекции, появившиеся в Java 2, представляют общую технологию хранения и доступа к объектам. Скорость обработки коллекций повысилась по сравнению с предыдущей версией языка за счет отказа от их потокобезопасности. Поэтому если объект коллекции может быть доступен из различных потоков, что наиболее естественно для распределенных приложений, следует использовать коллекции из Java 1.

Так как в коллекциях при практическом программировании хранится набор ссылок на объекты одного типа, следует обезопасить коллекцию от появления ссылок на другие не разрешенные логикой приложения типы. Такие ошибки при использовании нетипизированных коллекций выявляются на стадии выполнения, что повышает трудозатраты на исправление и верификацию кода. Поэтому начиная с версии 5.0 коллекции стали типизированными.

Более удобным стал механизм работы с коллекциями, а именно:

- предварительное сообщение компилятору о типе ссылок, которые будут храниться в коллекции, при этом проверка осуществляется на этапе компиляции;
- отсутствие необходимости постоянно преобразовывать возвращаемые по ссылке объекты (тип **Object**) к требуемому типу.

Структура коллекций характеризует способ, с помощью которого программы Java обрабатывают группы объектов. Так как **Object** – суперкласс для всех классов, то в коллекции можно хранить объекты любого типа, кроме базовых. Коллекции – это динамические массивы, связные списки, деревья, множества, хэш-таблицы, стеки, очереди.

Интерфейсы коллекций:

Map<K,V> – карта отображения вида "ключ-значение";

Collection<E> – вершина иерархии остальных коллекций;

List<E> – специализирует коллекции для обработки списков;

Set<E> – специализирует коллекции для обработки множеств, содержащих уникальные элементы.

Все классы коллекций реализуют также интерфейсы **Serializable**, **Cloneable** (кроме **WeakHashMap**). Кроме того, классы, реализующие интерфейсы **List<E>** и **Set<E>**, реализуют также интерфейс **Iterable<E>**.

В интерфейсе **Collection<E>** определены методы, которые работают на всех коллекциях:

boolean add(E obj) – добавляет **obj** к вызывающей коллекции и возвращает **true**, если объект добавлен, и **false**, если **obj** уже элемент коллекции;

boolean addAll(Collection<? extends E> c) – добавляет все элементы коллекции к вызывающей коллекции;

void clear() – удаляет все элементы из коллекции;

boolean contains(Object obj) – возвращает **true**, если вызывающая коллекция содержит элемент **obj**;

boolean equals(Object obj) – возвращает **true**, если коллекции эквивалентны;

boolean isEmpty() – возвращает **true**, если коллекция пуста;

Iterator<E> iterator() – извлекает итератор;

boolean remove(Object obj) – удаляет **obj** из коллекции;

int size() – возвращает количество элементов в коллекции;

Object[] toArray() – копирует элементы коллекции в массив объектов;

<T> T[] toArray(T a[]) – копирует элементы коллекции в массив объектов определенного типа.

Для работы с элементами коллекции применяются следующие интерфейсы:

Comparator<T> – для сравнения объектов;

Iterator<E>, **ListIterator<E>**, **Map.Entry<K,V>** – для перечисления и доступа к объектам коллекции.

Интерфейс **Iterator<E>** используется для построения объектов, которые обеспечивают доступ к элементам коллекции. К этому типу относится объект, возвращаемый методом **iterator()**. Такой объект позволяет просматривать

содержимое коллекции последовательно, элемента за элементом. Позиции итератора располагаются в коллекции между элементами. В коллекции, состоящей из N элементов, существует N+1 позиций итератора.

Методы интерфейса **`Iterator<E>:`**

`boolean hasNext()` – проверяет наличие следующего элемента, а в случае его отсутствия (завершения коллекции) возвращает **`false`**. Итератор при этом остается неизменным;

`E next()` – возвращает объект, на который указывает итератор, и передвигает текущий указатель на следующий, предоставляя доступ к следующему элементу. Если следующий элемент коллекции отсутствует, то метод **`next()`** генерирует исключение **`NoSuchElementException`**;

`void remove()` – удаляет объект, возвращенный последним вызовом метода **`next()`**.

Интерфейс **`ListIterator<E>`** расширяет интерфейс **`Iterator<E>`** и предназначен в основном для работы со списками. Наличие методов **`E previous()`**, **`int previousIndex()`** и **`boolean hasPrevious()`** обеспечивает обратную навигацию по списку. Метод **`int nextIndex()`** возвращает номер следующего итератора. Метод **`void add(E obj)`** позволяет вставлять элемент в список текущей позиции. Вызов метода **`void set(E obj)`** производит замену текущего элемента списка на объект, передаваемый методу в качестве параметра.

Интерфейс **`Map.Entry`** предназначен для извлечения ключей и значений карты с помощью методов **`K getKey()`** и **`V getValue()`** соответственно. Вызов метода **`V setValue(V value)`** заменяет значение, ассоциированное с текущим ключом.

Списки

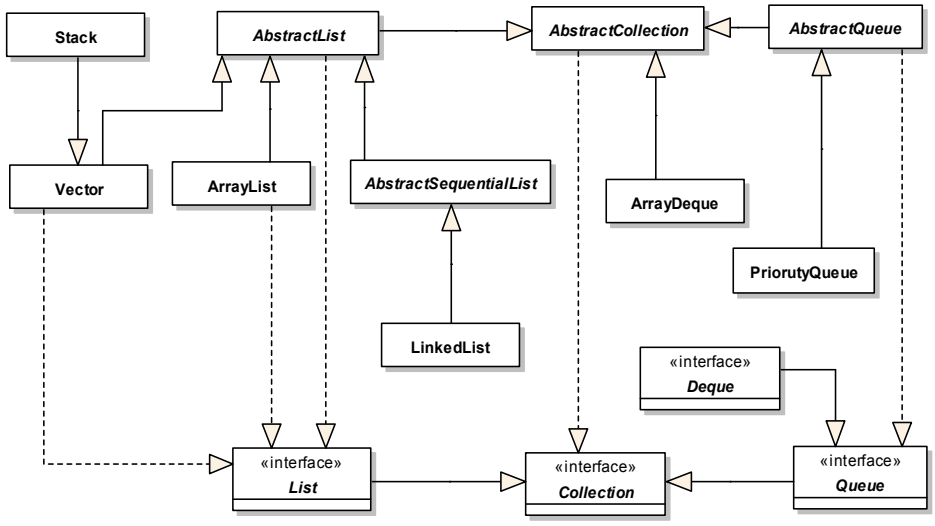

Рис. 10.1. Иерархия наследования списков

Класс **ArrayList<E>** – динамический массив объектных ссылок. Расширяет класс **AbstractList<E>** и реализует интерфейс **List<E>**. Класс имеет конструкторы:

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

Практически все методы класса являются реализацией абстрактных методов из суперклассов и интерфейсов. Методы интерфейса **List<E>** позволяют вставлять и удалять элементы из позиций, указываемых через отсчитываемый от нуля индекс:

void add(int index, E element) – вставляет **element** в позицию, указанную в **index**;

void addAll(int index, Collection<? extends E> c) – вставляет в вызывающий список все элементы коллекции **c**, начиная с позиции **index**;

E get(int index) – возвращает элемент в виде объекта из позиции **index**;

int indexOf(Object ob) – возвращает индекс указанного объекта;

E remove(int index) – удаляет объект из позиции **index**;

E set(int index, E element) – заменяет объект в позиции **index**, возвращает при этом удаляемый элемент;

List<E> subList(int fromIndex, int toIndex) – извлекает часть коллекции в указанных границах.

Удаление и добавление элементов для такой коллекции представляет собой ресурсоемкую задачу, поэтому объект **ArrayList<E>** лучше всего подходит для хранения неизменяемых списков.

```
/* пример #1 : создание параметризованной коллекции : DemoGeneric.java */
package chapt10;
import java.util.*;

public class DemoGeneric {
     public static void main(String args[]) {
          ArrayList<String> list = new ArrayList<String>();
          // ArrayList<int> b = new ArrayList<int>(); // ошибка компиляции
          list.add("Java");
          list.add("Fortress");
          String res = list.get(0);/* компилятор "знает"
                                      тип значения */
          // list.add(new StringBuilder("C#") ); // ошибка компиляции
          // компилятор не позволит добавить "посторонний" тип
          System.out.print(list);
     }
}
```

В результате будет выведено:

```
[Java, Fortress]
```

В данной ситуации не создается новый класс для каждого конкретного типа и сама коллекция не меняется, просто компилятор снабжается информацией о типе элементов, которые могут храниться в **list**. При этом параметром коллекции может быть только объектный тип.

Следует отметить, что указывать тип следует при создании ссылки, иначе будет позволено добавлять объекты всех типов.

```
/* пример # 2 : некорректная коллекция : UncheckCheck.java */
package chapt10;
import java.util.*;

public class UncheckCheck {
     public static void main(String args[]) {
          ArrayList list = new ArrayList();
          list.add(71);
          list.add(new Boolean("TruE"));
          list.add("Java 1.6.0");

          // требуется приведение типов
            int i = (Integer)list.get(0);
            boolean b = (Boolean)list.get(1);
            String str = (String)list.get(2);
          for (Object ob : list)
            System.out.println("list " + ob);

          ArrayList<Integer> s = new ArrayList<Integer>();
          s.add(71);
          s.add(92);
          // s.add("101");// ошибка компиляции: s параметризован
          for (Integer ob : s)
             System.out.print("int " + ob);
     }
}
```

В результате будет выведено:

```
list 71
list true
list Java 1.6.0
int 71
int 92
```

Чтобы параметризация коллекции была полной, необходимо указывать параметр и при объявлении ссылки, и при создании объекта.

Объект типа **Iterator** может использоваться для последовательного перебора элементов коллекции. Ниже приведен пример заполнения списка псевдослучайными числами, подсчет с помощью итератора количества положительных и удаление из списка неположительных значений.

```
/* пример # 3 : работа со списком : DemoIterator.java */
package chapt10;
import java.util.*;
```

```
public class DemoIterator {
      public static void main(String[] args) {
            ArrayList<Double> c =
                  new ArrayList<Double>(7);
            for(int i = 0 ;i < 10; i++) {
                  double z = new Random().nextGaussian();
                  c.add(z);//заполнение списка
            }
            //вывод списка на консоль
            for(Double d: c) {
            System.out.printf("%.2f ",d);
            }
            int positiveNum = 0;
            int size = c.size();//определение размера коллекции

            //извлечение итератора
            Iterator<Double> it = c.iterator();

      //проверка существования следующего элемента
      while(it.hasNext()) {
      //извлечение текущего элемента и переход к следующему
       if (it.next() > 0) positiveNum++;
       else it.remove();//удаление неположительного элемента
                        }
    System.out.printf("%nКоличество положительных: %d ",
                                          positiveNum);
    System.out.printf("%nКоличество неположительных: %d ",
                              size - positiveNum);
    System.out.println("\nПоложительная коллекция");
            for(Double d : c) {
                  System.out.printf("%.2f ",d);
                  }
      }
}
```

В результате на консоль будет выведено:

```
0,69 0,33 0,51 -1,24 0,07 0,46 0,56 1,26 -0,84 -0,53
Количество положительных: 7
Количество отрицательных: 3
Положительная коллекция
0,69 0,33 0,51 0,07 0,46 0,56 1,26
```

Для доступа к элементам списка может также использоваться интерфейс **ListIterator<E>**, который позволяет получить доступ сразу в необходимую программисту позицию списка. Такой способ доступа возможен только для списков.

```
/* пример # 4 : замена, удаление и поиск элементов : DemoListMethods.java */
package chapt10;
import java.util.*;
```

```
public class DemoListMethods {
      public static void main(String[] args) {
            ArrayList<Character> a =
                  new ArrayList<Character>(5);
            System.out.println("коллекция пуста: "
                                          + a.isEmpty());
            for (char c = 'a'; c < 'h'; ++c) {
                  a.add(c);
            }
            char ch = 'a';
            a.add(6, ch); // заменить 6 на >=8 – ошибка выполнения
            System.out.println(a);
      ListIterator<Character> it;// параметризация обязательна
      it= a.listIterator(2);// извлечение итератора списка в позицию
      System.out.println("добавление элемента в позицию "
                                    + it.nextIndex());
      it.add('X');// добавление элемента без замены в позицию итератора
      System.out.println(a);
            // сравнить методы
      int index = a.lastIndexOf(ch); // a.indexOf(ch);
      a.set(index, 'W'); // замена элемента без итератора
      System.out.println(a + "после замены элемента");
            if (a.contains(ch)) {
                  a.remove(a.indexOf(ch));
                  }
            System.out.println(a + "удален элемент " + ch);
      }
}
```

В результате будет выведено:

```
коллекция пуста: true
[a, b, c, d, e, f, a, g]
добавление элемента в позицию 2
[a, b, X, c, d, e, f, a, g]
[a, b, X, c, d, e, f, W, g]после замены элемента
[b, X, c, d, e, f, W, g]удален элемент a
```

Коллекция **LinkedList<E>** реализует связанный список. В отличие от массива, который хранит объекты в последовательных ячейках памяти, связанный список хранит объекты отдельно, но вместе со ссылками на следующее и предыдущее звенья последовательности.

В добавление ко всем имеющимся методам в **LinkedList<E>** реализованы методы **void addFirst(E ob)**, **void addLast(E ob)**, **E getFirst()**, **E getLast()**, **E removeFirst()**, **E removeLast()** добавляющие, извлекающие, удаляющие и извлекающие первый и последний элементы списка соответственно.

Класс **LinkedList<E>** реализует интерфейс **Queue<E>**, что позволяет предположить, что такому списку легко придать свойства очереди. К тому же специализированные методы интерфейса **Queue<E>** по манипуляции первым и

последним элементами такого списка **E element()**, **boolean offer(E o)**, **E peek()**, **E poll()**, **E remove()** работают немного быстрее, чем соответствующие методы класса **LinkedList<E>**.

Методы интерфейса **Queue<E>**:

E element() – возвращает, но не удаляет головной элемент очереди;

boolean offer(E o) – вставляет элемент в очередь, если возможно;

E peek() – возвращает, но не удаляет головной элемент очереди, возвращает **null**, если очередь пуста;

E poll() – возвращает и удаляет головной элемент очереди, возвращает **null**, если очередь пуста;

E remove() – возвращает и удаляет головной элемент очереди.

Методы **element()** и **remove()** отличаются от методов **peek()** и **poll()** тем, что генерируют исключение, если очередь пуста.

```
/* пример # 5 : добавление и удаление элементов : DemoLinkedList.java */
package chapt10;
import java.util.*;

public class DemoLinkedList {
     public static void main(String[] args){
          LinkedList<Number> a = new LinkedList<Number>();
          for(int i = 10; i <= 15; i++)
             a.add(i);
          for(int i = 16; i <= 20; i++)
             a.add(new Float(i));
          ListIterator<Number> list = a.listIterator(10);
          System.out.println("\n"+ list.nextIndex()
                                        + "-й индекс");
          list.next(); // важно!
          System.out.println(list.nextIndex()
                                        + "-й индекс");
          list.remove(); // удаление элемента с текущим индексом
          while(list.hasPrevious())
             System.out.print(list.previous()+"  "); /*вывод
                                          в обратном порядке*/
          // демонстрация работы методов
          a.removeFirst();
          a.offer(71); // добавление элемента в конец списка
          a.poll(); // удаление нулевого элемента из списка
          a.remove(); // удаление нулевого элемента из списка
          a.remove(1); // удаление первого элемента из списка
          System.out.println("\n" + a);

          Queue<Number> q = a; // список в очередь
          for (Number i : q) // вывод элементов
             System.out.print(i + " ");
          System.out.println(" :size= " + q.size());

          // удаление пяти элементов
```

```
        for (int i = 0; i < 5; i++) {
            Number res = q.poll();
        }
        System.out.print("size= " + q.size());
    }
}
```

В результате будет выведено:

```
10-й индекс
11-й индекс
19.0  18.0  17.0  16.0  15  14  13  12  11  10
[13, 15, 16.0, 17.0, 18.0, 19.0, 71]
13 15 16.0 17.0 18.0 19.0 71  :size= 7
size= 2
```

При реализации интерфейса **Comparator<T>** существует возможность сортировки списка объектов конкретного типа по правилам, определенным для этого типа. Для этого необходимо реализовать метод **int compare(T ob1, T ob2)**, принимающий в качестве параметров два объекта для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки. Этот метод автоматически вызывается методом **public static <T> void sort(List<T> list, Comparator<? super T> c)** класса **Collections**, в качестве первого параметра принимающий коллекцию, в качестве второго – объект-comparator, из которого извлекается правило сортировки.

```
/* пример # 6 : авторская сортировка списка: UniqSortMark.java */
package chapt10;
import java.util.Comparator;

public class Student implements Comparator<Student> {
    private int idStudent;
    private float meanMark;

    public Student(float m, int id) {
        meanMark = m;
        idStudent = id;
    }
    public Student() {
    }
    public float getMark() {
        return meanMark;
    }
    public int getIdStudent() {
        return idStudent;
    }
    // правило сортировки
    public int compare(Student one, Student two) {
        return
        (int)(Math.ceil(two.getMark() - one.getMark()));
    }
}
```

```
package chapt10;
import java.util.*;

public class UniqSortMark {
     public static void main(String[] args) {
          ArrayList<Student> p = new ArrayList<Student>();
          p.add(new Student(3.9f, 52201));
          p.add(new Student(3.65f, 52214));
          p.add(new Student(3.71f, 52251));
          p.add(new Student(3.02f, 52277));
          p.add(new Student(3.81f, 52292));
          p.add(new Student(9.55f, 52271));
          // сортировка списка объектов
          try {
       Collections.sort(p, Student.class.newInstance());
          } catch (InstantiationException e1) {
             //невозможно создать объект класса
             e1.printStackTrace();
          } catch (IllegalAccessException e2) {
             e2.printStackTrace();
          }
          for (Student ob : p)
               System.out.printf("%.2f ", ob.getMark());
     }
}
```

В результате будет выведено:

```
9,55 3,90 3,81 3,71 3,65 3,02
```

Метод **boolean equals(Object obj)** интерфейса **Comparator<T>**, который обязан выполнять свой контракт, возвращает **true** только в случае если соответствующий метод **compare()** возвращает **0**.

Для создания возможности сортировки по другому полю **id** класса **Student** следует создать новый класс, реализующий **Comparator** по новым правилам.

```
/* пример # 7 : другое  правило сортировки: StudentId.java */
package chapt10;

public class StudentId implements Comparator<Student> {
     public int compare(Student one, Student two) {
          return two.getIdStudent() - one.getIdStudent();
     }
   }
```

При необходимости сортировки по полю **id** в качестве второго параметра следует объект класса **StudentId**:

```
Collections.sort(p, StudentId.class.newInstance());
```

Параметризация коллекций позволяет разрабатывать безопасные алгоритмы, создание которых потребовало бы несколько больших затрат в предыдущих версиях языка.

Deque

Интерфейс **Deque** определяет «двунаправленную» очередь и, соответственно, методы доступа к первому и последнему элементам двусторонней очереди. Методы обеспечивают удаление, вставку и обработку элементов. Каждый из этих методов существует в двух формах. Одни методы создают исключительную ситуацию в случае неудачного завершения, другие возвращают какое-либо из значений (**null** или **false** в зависимости от типа операции). Вторая форма добавления элементов в очередь сделана специально для реализаций **Deque**, имеющих ограничение по размеру. В большинстве реализаций операции добавления заканчиваются успешно.

В следующим примере реализована работа с интерфейсом **Deque**. Методы **addFirst()**, **addLast()** вставляют элементы в начало и в конец очереди соответственно. Метод **add()** унаследован от интерфейса **Queue** и абсолютно аналогичен методу **addLast()** интерфейса **Deque**.

```
/* пример # 8 : демонстрация Deque : DequeRunner.java */
package chapt10;
import java.util.*;

public class DequeRunner {
      public static void printDeque(Deque<?> d){
                  for (Object de : d)
                        System.out.println(de + "; ");
            }
      public static void main(String[] args) {
            Deque<String> deque = new ArrayDeque<String>();
                  deque.add(new String("5"));
                  deque.addFirst("A");
                  //deque.addLast(new Integer(5));//ошибка компиляции
                  System.out.println(deque.peek());
                  System.out.println("Before:");
                  printDeque(deque);
                  deque.pollFirst();
                  System.out.println(deque.remove(5));
                  System.out.println("After:");
                  printDeque(deque);
      }
}
```

В результате на консоль будет выведено:

```
A
Before:
A;
5;
false
After:
5;
```

Множества

Интерфейс **Set<E>** объявляет поведение коллекции, не допускающей дублирования элементов. Интерфейс **SortedSet<E>** наследует **Set<E>** и объявляет поведение набора, отсортированного в возрастающем порядке, заранее определенном для класса. Интерфейс **NavigableSet** существенно облегчает поиск элементов.

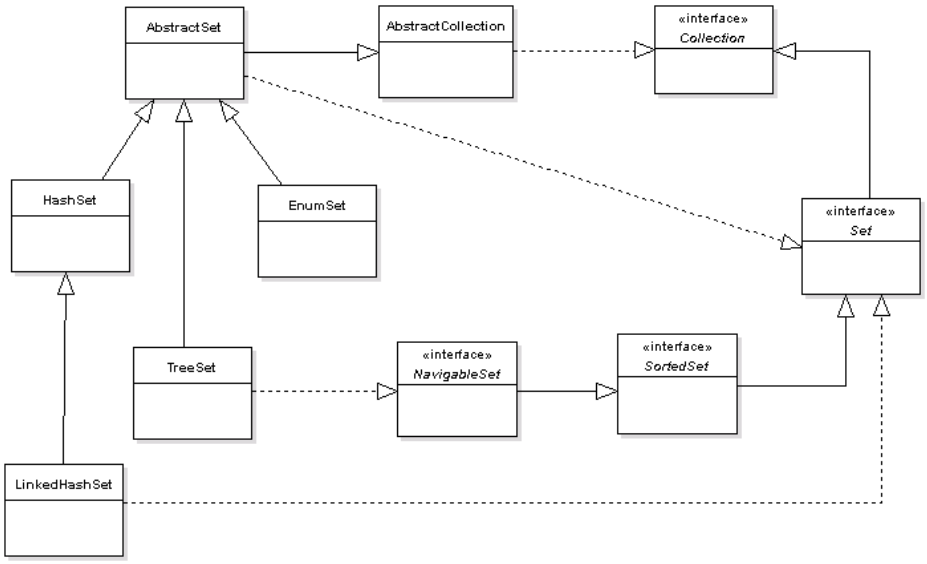

Рис. 10.2. Иерархия наследования множеств

Класс **HashSet<E>** наследуется от абстрактного суперкласса **AbstractSet<E>** и реализует интерфейс **Set<E>**, используя хэш-таблицу для хранения коллекции. Ключ (хэш-код) используется вместо индекса для доступа к данным, что значительно ускоряет поиск определенного элемента. Скорость поиска существенна для коллекций с большим количеством элементов. Все элементы такого множества упорядочены посредством хэш-таблицы, в которой хранятся хэш-коды элементов.

Конструкторы класса:

HashSet()

HashSet(Collection <? extends E> c)

HashSet(int capacity)

HashSet(int capacity, float loadFactor), где **capacity** – число ячеек для хранения хэш-кодов.

/ пример # 9 : использование множества для вывода всех уникальных слов из файла : DemoHashSet.java */*

```
package chapt10;
import java.util.*;
import java.io.*;
```

```
public class DemoHashSet {
     public static void main(String[] args) {
     HashSet<String> words = new HashSet<String>(100);
     // использовать коллекции LinkedHashSet или TreeSet
     long callTime = System.nanoTime();
     try {
            BufferedReader in =
                  new BufferedReader(
                        new FileReader("c://pushkin.txt"));
              String line = "";
       while ((line = in.readLine()) != null) {
         StringTokenizer tokenizer =
               new StringTokenizer(line,
                     " (){}[]<>#*!?.,:;-\'\"/");
               while (tokenizer.hasMoreTokens()) {
                    String word = tokenizer.nextToken();
                    words.add(word.toLowerCase());
               }
        }
     } catch (IOException e) {
         System.err.println(e);
     }
     Iterator<String> it = words.iterator();
     while (it.hasNext())
         System.out.println(it.next());
     long totalTime = System.nanoTime()- callTime;
     System.out.println("различных слов: " + words.size()
                  + ", " + totalTime + " наносекунд");
     }
}
```

Класс **TreeSet<E>** для хранения объектов использует бинарное дерево. При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что все добавляемые элементы реализуют интерфейсы **Comparator** и **Comparable**. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

Конструкторы класса:

TreeSet()
TreeSet(Collection <? extends E> c)
TreeSet(Comparator <? super E> c)
TreeSet(SortedSet <E> s)

Класс **TreeSet<E>** содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов **E first()** и **E last()**. Методы **SortedSet<E> subSet(E from, E to)**, **SortedSet<E> tailSet(E from)** и **SortedSet<E> headSet(E to)** предназначены для извлечения определенной части множества. Метод **Comparator <? super E> comparator()** возвращает объект

Comparator, используемый для сортировки объектов множества или **null**, если выполняется обычная сортировка.

```
/* пример # 10: создание множества из списка и его методы: DemoTreeSet.java */
package chapt10;
import java.util.*;

public class DemoTreeSet {
     public static void main(String[] args) {
          ArrayList<String> c = new ArrayList<String>();
          boolean b;
          for (int i = 0; i < 6; i++)
               c.add((int) (Math.random() * 71) + "Y ");
          System.out.println(c + "список");
          TreeSet<String> set = new TreeSet<String>(c);
          System.out.println(set + "множество");
          b = set.add("5 Element"); // добавление(b=true)
          b = set.add("5 Element"); // добавление(b=false)

          // после добавления
          System.out.println(set + "add");
          System.out.println(set.comparator()); //null !!!

          // извлечение наибольшего и наименьшего элементов
          System.out.println(set.last() + " "
                    + set.first());
     }
}
```

В результате может быть выведено:

```
[44Y , 56Y , 49Y , 26Y , 49Y , 2Y ]список
[2Y , 26Y , 44Y , 49Y , 56Y ]множество
[2Y , 26Y , 44Y , 49Y , 5 Element, 56Y ]add
null
56Y  2Y
```

Множество инициализируется списком и сортируется сразу же в процессе создания. После добавления нового элемента производится неудачная попытка добавить его повторно. С помощью итератора элемент может быть найден и удален из множества. Для множества, состоящего из обычных строк, используется по умолчанию правило обычной лексикографической сортировки, поэтому метод **comparator()** возвращает **null**.

Если попытаться заменить тип **String** на **StringBuilder** или **StringBuffer**, то создать множество **TreeSet** так просто не удастся. Решением такой задачи будет создание нового класса с полем типа **StringBuilder** и реализацией интерфейса **Comparable<T>** вида:

```
/* пример # 11 :пользовательский класс, объект которого может быть добавлен
в множество TreeSet : Message.java */
package chapt10;
import java.util.*;
```

```
public class Message implements Comparable<Message> {
      private StringBuilder str;
      private int idSender;

      public Message(StringBuilder s, int id) {
            super();
            this.str = s;
            idSender = id;
      }
      public String getStr() {
            return str.toString();
      }
      public int getId() {
            return idSender;
      }
      public int compareTo(Message a0) {
            return (idSender - a0.getId());
      }
}
```

Предлагаемое решение универсально для любых пользовательских типов.

Абстрактный класс **EnumSet<E extends Enum<E>>** наследуется от абстрактного класса **AbstractSet**. Специально реализован для работы с типами **enum**. Все элементы такой коллекции должны принадлежать единственному типу **enum**, определенному явно или неявно. Внутренне множество представимо в виде вектора битов, обычно единственного **long**. Множества нумераторов поддерживают перебор по диапазону из нумераторов. Скорость выполнения операций над таким множеством очень высока, даже если в ней участвует большое количество элементов.

Создать объект этого класса можно только с помощью статических методов. Метод **EnumSet<E> noneOf(Class<E> elemType)** создает пустое множество нумерованных констант с указанным типом элемента, метод **allOf(Class<E> elementType)** создает множество нумерованных констант, содержащее все элементы указанного типа. Метод **of(E first, E... rest)** создает множество, первоначально содержащее указанные элементы. С помощью метода **complementOf(EnumSet<E> s)** создается множество, содержащее все элементы, которые отсутствуют в указанном множестве. Метод **range(E from, E to)** создает множество из элементов, содержащихся в диапазоне, определенном двумя элементами. При передаче вышеуказанным методам в качестве параметра **null** будет сгенерирована исключительная ситуация **NullPointerException**.

```
/* пример # 12 : использование множества enum-типов : UseEnumSet.java */
package chapt10;
import java.util.EnumSet;

enum Faculty{ FFSM, MMF, FPMI, FMO, GEO }

public class UseEnumSet {
```

```
      public static void main(String[] args) {
      /*множество set1 содержит элементы типа enum из интервала,
       определенного двумя элементами*/
            EnumSet <Faculty> set1 =
                  EnumSet.range(Faculty.MMF, Faculty.FMO);
      /*множество set2 будет содержать все элементы, не содержащиеся
      в множестве set1*/
            EnumSet <Faculty> set2 =
                  EnumSet.complementOf(set1);
            System.out.println(set1);
            System.out.println(set2);
      }
}
```

В результате будет выведено:

```
[MMF, FPMI, FMO]
[FFSM, GEO]
```

В следующем примере показано использование интерфейса **NavigableSet**. Метод **first()** возвращает первый элемент из множества. Метод **subSet(E fromElement, E toElement)** возвращает список элементов, находящихся между **fromElement** и **toElement**, причем последний не включается. Методы **headSet(E element)** и **tailSet(E element, boolean inclusive)** возвращают то множество элементов, которое меньше либо больше **element** соответственно. Если **inclusive** равно **true**, то элемент включается в найденное множество и не включается в противном случае.

```
/* пример # 13 : использование множества NavigableSet: NavigableSetTest.java */
package chapt10;
import java.util.*;

public class NavigableSetTest {
  public static void main(String[] args) {
     HashSet<String> city = new HashSet<String>();
     city.add("Minsk");
     city.add("Moscow");
     city.add("Polotsk");
     city.add("Brest");
     NavigableSet<String> ns = new TreeSet<String>(city);
     System.out.println("All: " + ns);
     System.out.println("First: " + ns.first());
     System.out.println("Between Minsk and Polotsk: "
                        + ns.subSet("Minsk","Polotsk"));
     System.out.println("Before Minsk: "
                              + ns.headSet("Minsk"));
     System.out.println("After Minsk: "
                              + ns.tailSet("Minsk",false));
  }
}
```

В результате на консоль будет выведено:

```
All: [Brest, Minsk, Moscow, Polotsk]
First: Brest
Between Minsk and Polotsk: [Minsk, Moscow]
Before Minsk: [Brest]
After Minsk: [Moscow, Polotsk]
```

Карты отображений

Карта отображений – это объект, который хранит пару "ключ-значение". Поиск объекта (значения) облегчается по сравнению с множествами за счет того, что его можно найти по его уникальному ключу. Уникальность объектов-ключей должна обеспечиваться переопределением методов **`hashCode()`** и **`equals()`** пользовательским классом. Если элемент с указанным ключом отсутствует в карте, то возвращается значение **`null`**.

Классы карт отображений:

`AbstractMap<K,V>` – реализует интерфейс **`Map<K,V>`**;

`HashMap<K,V>` – расширяет **`AbstractMap<K,V>`**, используя хэш-таблицу, в которой ключи отсортированы относительно значений их хэш-кодов;

`TreeMap<K,V>` – расширяет **`AbstractMap<K,V>`**, используя дерево, где ключи расположены в виде дерева поиска в строгом порядке.

`WeakHashMap<K,V>` позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости приложения.

`LinkedHashMap<K,V>` запоминает порядок добавления объектов в карту и образует при этом дважды связанный список ключей. Этот механизм эффективен, только если превышен коэффициент загруженности карты при работе с кэш-памятью и др.

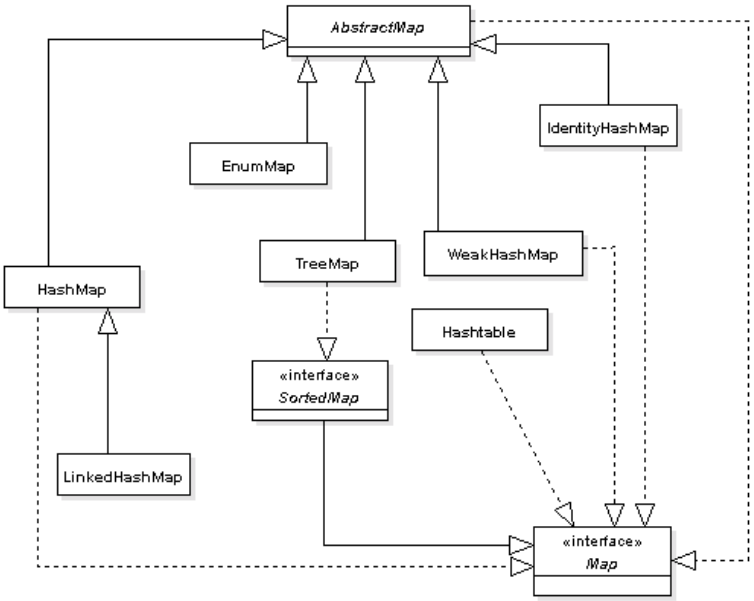

Рис. 10.3. Иерархия наследования карт

Для класса **IdentityHashMap<K,V>** хэш-коды объектов-ключей вычисляются методом **System.identityHashCode()** по адресу объекта в памяти, в отличие от обычного значения **hashCode()**, вычисляемого сугубо по содержимому самого объекта.

Интерфейсы карт:

Map<K,V> – отображает уникальные ключи и значения;

Map.Entry<K,V> – описывает пару "ключ-значение";

SortedMap<K,V> – содержит отсортированные ключи и значения;

NavigableMap<K,V> – добавляет новые возможности поиска по ключу.

Интерфейс **Map<K,V>** содержит следующие методы:

void clear() – удаляет все пары из вызываемой карты;

boolean containsKey(Object key) – возвращает **true**, если вызывающая карта содержит **key** как ключ;

boolean containsValue(Object value) – возвращает **true**, если вызывающая карта содержит **value** как значение;

Set<Map.Entry<K,V>> entrySet() – возвращает множество, содержащее значения карты;

Set<K> keySet() – возвращает множество ключей;

V get(Object obj) – возвращает значение, связанное с ключом **obj**;

V put(K key, V value) – помещает ключ **key** и значение **value** в вызывающую карту. При добавлении в карту элемента с существующим ключом произойдет замена текущего элемента новым. При этом метод возвратит заменяемый элемент;

void putAll(Map <? extends K, ? extends V> t) – помещает коллекцию **t** в вызывающую карту;

V remove(Object key) – удаляет пару "ключ-значение" по ключу **key**;

Collection<V> values() – возвращает коллекцию, содержащую значения карты.

Интерфейс **Map.Entry<K,V>** содержит следующие методы:

K getKey() – возвращает ключ текущего входа;

V getValue() – возвращает значение текущего входа;

V setValue(V obj) – устанавливает значение объекта **obj** в текущем входе.

В примере показаны способы создания хэш-карты и доступа к ее элементам.

```
/* пример # 14 : создание хэш-карты и замена элемента по ключу:
DemoHashMap.java */
package chapt10;
import java.util.*;

public class DemoHashMap {
     public static void main(String[] args){
          HashMap<Integer, String> hm =
                    new HashMap<Integer, String>(5);
          for (int i = 11; i < 15; i++)
                    hm.put(i, i + "EL");
```

```
        System.out.println(hm);
        hm.put(12, "14EL");
        System.out.println(hm + "с заменой элемента");
        String a = hm.get(12);
        System.out.println(a + " - найден по ключу '12'");
        /* вывод хэш-таблицы с помощью методов интерфейса
         Map.Entry<K,V> */
        Set<Map.Entry<Integer, String>> setvalue =
                                        hm.entrySet();
        System.out.println(setvalue);
        Iterator<Map.Entry<Integer, String>> i =
                        setvalue.iterator();
        while (i.hasNext()) {
             Map.Entry<Integer, String> me = i.next();
             System.out.print(me.getKey()+" : ");
             System.out.println(me.getValue());
        }
    }
}
```

```
{13=13EL, 14=14EL, 12=12EL, 11=11EL}
{13=13EL, 14=14EL, 12=14EL, 11=11EL}с заменой элемента
14EL - найден по ключу '12'
[13=13EL, 14=14EL, 12=14EL, 11=11EL]
13 : 13EL
14 : 14EL
12 : 14EL
11 : 11EL
```

Ниже приведен фрагмент кода корпоративной системы, где продемонстрированы возможности класса **HashMap<K,V>** и интерфейса **Map.Entry<K,V>** при определении прав пользователей.

```
/* пример # 15 : применение коллекций при проверке доступа в систему :
DemoSecurity.java */
package chapt10;
import java.util.*;

public class DemoSecurity {
        public static void main(String[] args) {
        CheckRight.startUsing(2041, "Артем");
        CheckRight.startUsing(2420, "Ярослав");
        /*
         *добавление еще одного пользователя и
         * проверка его на возможность доступа
         */
        CheckRight.startUsing(2437, "Анастасия");
        CheckRight.startUsing(2041, "Артем");
    }
}
```

```
/* пример # 16 : класс проверки доступа в систему: CheckRight.java */
package chapt10;
import java.util.*;

public class CheckRight {
     private static HashMap<Integer, String> map =
                         new HashMap<Integer, String> ();

     public static void startUsing(int id, String name) {
          if (canUse(id)){
               map.put(id, name);
               System.out.println("доступ разрешен");
          } else {
               System.out.println("в доступе отказано");
          }
     }
     public static boolean canUse(int id) {
          final int MAX_NUM = 2; // заменить 2 на 3
          int currNum = 0;
          if (!map.containsKey(id))
               currNum = map.size();
          return currNum < MAX_NUM;
     }
}
```

В результате будет выведено:

```
доступ разрешен
доступ разрешен
в доступе отказано
доступ разрешен,
```

так как доступ в систему разрешен одновременно только для двух пользователей. Если в коде изменить значение константы **MAX_NUM** на большее, чем 2, то новый пользователь получит права доступа.

Класс **EnumMap<K extends Enum<K>, V>** в качестве ключа может принимать только объекты, принадлежащие одному типу **enum**, который должен быть определен при создании коллекции. Специально организован для обеспечения максимальной скорости доступа к элементам коллекции.

```
/* пример # 17 : пример работы с классом EnumMap: UseEnumMap.java */
package chapt10;
import java.util.EnumMap;

enum User {
      STUDENT, TUTOR, INSTRUCTOR, DEAN
}
class UserPriority {
      private int priority;

      public UserPriority(User k) {
            switch (k) {
```

```
            case STUDENT:
                  priority = 1; break;
            case TUTOR:
                  priority = 3; break;
            case INSTRUCTOR:
                  priority = 7; break;
            case DEAN:
                  priority = 10; break;
            default:
                  priority = 0;
            }
      }
      public int getPriority() {
            return priority;
      }
}
public class UseEnumMap {
      public static void main(String[] args) {
            EnumMap<User, UserPriority> faculty =
new EnumMap<User, UserPriority> (User.class);
            for (User user : User.values()) {
                  faculty.put(user,
                        new UserPriority(user));
            }
            for (User user : User.values()) {
                  System.out.println(user.name()
                        + "-> Priority:" +
((UserPriority) faculty.get(user)).getPriority());
            }
      }
}
```

В результате будет выведено:

```
STUDENT-> Priority:1
TUTOR-> Priority:3
INSTRUCTOR-> Priority:7
DEAN-> Priority:10
```

Унаследованные коллекции

В ряде распределенных приложений, например с использованием сервлетов, до сих пор применяются коллекции, более медленные в обработке, но при этом потокобезопасные, существовавшие в языке Java с момента его создания, а именно карта **Hashtable<K,V>**, список **Vector<E>** и перечисление **Enumeration<E>**. Все они также были параметризованы, но сохранили все свои особенности.

Класс **Hashtable<K,V>** реализует интерфейс **Map**, но обладает также несколькими интересными методами:

Enumeration<V> elements() – возвращает перечисление (аналог итератора) для значений карты;

Enumeration<K> keys() – возвращает перечисление для ключей карты.

```
/* пример # 18 : создание хэш-таблицы и поиск элемента по ключу:
  HashTableDemo.java */
package chapt10;
import java.util.*;
import java.io.*;

public class HashTableDemo {
     public static void main(String[] args) {
          Hashtable<Integer, Double> ht =
                         new Hashtable<Integer, Double>();
          for (int i = 0; i < 5; i++)
               ht.put(i, Math.atan(i));
          Enumeration<Integer> ek = ht.keys();
          int key;
          while (ek.hasMoreElements()) {
               key = ek.nextElement();
               System.out.printf("%4d  ", key);
          }
             System.out.println("");
          Enumeration<Double> ev = ht.elements();
          double value;
          while (ev.hasMoreElements()) {
               value = ev.nextElement();
               System.out.printf("%.2f  ", value);
          }
     }
}
```

В результате в консоль будет выведено:

```
   4     3     2     1     0
1,33  1,25  1,11  0,79  0,00
```

Принципы работы с коллекциями, в отличие от их структуры, со сменой версий языка существенно не изменились.

Класс Collections

Класс **Collections** содержит большое количество статических методов, предназначенных для манипулирования коллекциями.

С применением предыдущих версий языка было разработано множество коллекций, в которых никаких проверок нет, следовательно, при их использовании нельзя гарантировать, что в коллекцию не будет помещен "посторонний" объект. Для этого в класс **Collections** был добавлен новый метод – **checkedCollection()**:

```
public static <E> Collection <E>
      checkedCollection(Collection<E> c, Class<E> type)
```

Этот метод создает коллекцию, проверяемую на этапе выполнения, то есть в случае добавления "постороннего" объекта генерируется исключение **ClassCastException**:

```
/* пример # 19 : проверяемая коллекция: SafeCollection.java  */
package chapt10;
import java.util.*;

public class SafeCollection {
     public static void main(String args[]) {
          Collection c = Collections.checkedCollection(
                         new HashSet<String>(), String.class);
          c.add("Java");
          c.add(7.0); // ошибка времени выполнения
     }
}
```

В этот же класс добавлен целый ряд методов, специализированных для проверки конкретных типов коллекций, а именно: **checkedList()**, **checkedSortedMap()**, **checkedMap()**, **checkedSortedSet()**, **checkedSet()**, а также:

<T> boolean addAll(Collection<? super T> c, T... a) – добавляет в параметризованную коллекцию соответствующие параметризации элементы;

<T> void copy(List<? super T> dest, List<? extends T> src) – копирует все элементы из одного списка в другой;

boolean disjoint(Collection<?> c1, Collection<?> c2) – возвращает **true**, если коллекции не содержат одинаковых элементов;

<T> List <T> emptyList(), **<K,V> Map <K,V> emptyMap()**, **<T> Set <T> emptySet()** – возвращают пустой список, карту отображения и множество соответственно;

<T> void fill(List<? super T> list, T obj) – заполняет список заданным элементом ;

int frequency(Collection<?> c, Object o) – возвращает количество вхождений в коллекцию заданного элемента;

<T extends Object & Comparable <? super T>> T max(Collection<? extends T> coll),

<T extends Object & Comparable <? super T>> T min(Collection<? extends T> coll) – возвращают минимальный и максимальный элемент соответственно;

<T> T max(Collection <? extends T> coll, Comparator<? super T> comp),

<T> T min(Collection<? extends T> coll, Comparator<? super T> comp) – возвращают минимальный и максимальный элемент соответственно, используя **Comparator** для сравнения;

<T> List <T> nCopies(int n, T o) – возвращает список из **n** заданных элементов;

<T> boolean replaceAll(List<T> list, T oldVal, T newVal) – заменяет все заданные элементы новыми;

void reverse(List<?> list) – "переворачивает" список;

void rotate(List<?> list, int distance) – сдвигает список циклически на заданное число элементов;

void shuffle(List<?> list) – перетасовывает элементы списка;

<T> Set <T> singleton(T o), **singletonList(T o)**, **singletonMap(K key, V value)** – создают множество, список и карту отображения, состоящие из одного элемента;

<T extends Comparable<? super T>> void sort(List<T> list),

<T> void sort(List<T> list, Comparator<? super T> c) – сортировка списка, естественным порядком и используя **Comparator** соответственно;

void swap(List<?> list, int i, int j) – меняет местами элементы списка стоящие на заданных позициях.

```
/* пример # 20 : методы класса Collections: CollectionsDemo.java:
MyComparator.java */
package chapt10;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionsDemo {
      public static void main(String[] args) {
            MyComparator<Integer> comp =
                  new MyComparator<Integer>();
            ArrayList<Integer> list =
                  new ArrayList<Integer>();

            Collections.addAll(list, 1, 2, 3, 4, 5);
            Collections.shuffle(list);
            print(list);
            Collections.sort(list, comp);
            print(list);
            Collections.reverse(list);
            print(list);
            Collections.rotate(list, 3);
            print(list);
            System.out.println("min: "
            + Collections.min(list, comp));
            System.out.println("max: "
            + Collections.max(list, comp));

            List<Integer> singl =
                  Collections.singletonList(71);
            print(singl);
            //singl.add(21);//ошибка времени выполнения
      }
      static void print(List<Integer> c) {
```

```
            for (int i : c)
                  System.out.print(i + " ");
            System.out.println();
      }
}
package chapt10;
import java.util.Comparator;

public class MyComparator<T> implements Comparator<Integer>
{
      public int compare(Integer n, Integer m) {
            return m.intValue() - n.intValue();
      }
}
```

В результате будет выведено:

```
4 3 5 1 2
5 4 3 2 1
1 2 3 4 5
3 4 5 1 2
min: 5
max: 1
71
```

Класс Arrays

В пакете **java.util** находится класс **Arrays**, который содержит методы манипулирования содержимым массива, а именно для поиска, заполнения, сравнения, преобразования в коллекцию и прочие:

int binarySearch(параметры**)** – перегруженный метод организации бинарного поиска значения в массивах примитивных и объектных типов. Возвращает позицию первого совпадения;

void fill(параметры**)** – перегруженный метод для заполнения массивов значениями различных типов и примитивами;

void sort(параметры**)** – перегруженный метод сортировки массива или его части с использованием интерфейса **Comparator** и без него;

static <T> T[] copyOf(T[] original, int newLength) – заполняет массив определенной длины, отбрасывая элементы или заполняя **null** при необходимости;

static <T> T[] copyOfRange(T[] original, int from, int to) – копирует заданную область массива в новый массив;

<T> List<T> asList(T... a) – метод, копирующий элементы массива в объект типа **List<T>**.

В качестве простого примера применения указанных методов можно привести следующий код.

```
/* пример # 21 : методы класса Arrays : ArraysEqualDemo.java */
package chapt10;
import java.util.*;
```

```
public class ArraysEqualDemo {
    public static void main(String[] args) {
        char m1[] = new char[3];
        char m2[] = { 'a', 'b', 'c' }, i;
        Arrays.fill(m1, 'a');
        System.out.print("массив m1:");
        for (i = 0; i < 3; i++)
            System.out.print(" " + m1[i]);
        m1[1] = 'b';
        m1[2] = 'c';
        //m1[2]='x'; // приведет к другому результату
        if (Arrays.equals(m1, m2))
            System.out.print("\nm1 и m2 эквивалентны");
        else
           System.out.print("\nm1 и m2 не эквивалентны");
        m1[0] = 'z';
        Arrays.sort(m1);
        System.out.print("\nмассив m1:");
        for (i = 0; i < 3; i++)
            System.out.print(" " + m1[i]);
        System.out.print(
                  "\n значение 'c' находится в позиции-"
                  + Arrays.binarySearch(m1, 'c'));
           Integer arr[] = {35, 71, 92};
           //вывод массива объектов в строку
           System.out.println(Arrays.deepToString(arr));
           //вычисление хэш-кода исходя и значений элементов
           System.out.println(Arrays.deepHashCode(arr));
           Integer arr2[] = {35, 71, 92};
           //сравнение массивов по седержимому
      System.out.println(Arrays.deepEquals(arr, arr2));
           char m3[] = new char[5];
           // копирование массива
           m3 = Arrays.copyOf(m1, 5);
           System.out.print("массив m3:");
           for (i = 0; i < 5; i++)
                 System.out.print(" " + m3[i]);
    }
}
```

В результате компиляции и запуска будет выведено:

```
массив m1: a a a
m1 и m2 эквивалентны
массив m1: b c z
значение 'c' находится в позиции 1
[35, 71, 92]
65719
true
массив m3: b c z □ □
```

Задания к главе 10

Вариант А

1. Ввести строки из файла, записать в список. Вывести строки в файл в обратном порядке.
2. Ввести число, занести его цифры в стек. Вывести число, у которого цифры идут в обратном порядке.
3. Создать в стеке индексный массив для быстрого доступа к записям в бинарном файле.
4. Создать список из элементов каталога и его подкаталогов.
5. Создать стек из номеров записи. Организовать прямой доступ к элементам записи.
6. Занести стихотворения одного автора в список. Провести сортировку по возрастанию длин строк.
7. Задать два стека, поменять информацию местами.
8. Определить множество на основе множества целых чисел. Создать методы для определения пересечения и объединения множеств.
9. Списки (стеки, очереди) I(1..n) и U(1..n) содержат результаты n-измерений тока и напряжения на неизвестном сопротивлении R. Найти приближенное число R методом наименьших квадратов.
10. С использованием множества выполнить попарное суммирование произвольного конечного ряда чисел по следующим правилам: на первом этапе суммируются попарно рядом стоящие числа, на втором этапе суммируются результаты первого этапа и т.д. до тех пор, пока не останется одно число.
11. Сложить два многочлена заданной степени, если коэффициенты многочленов хранятся в объекте **`HashMap`**.
12. Умножить два многочлена заданной степени, если коэффициенты многочленов хранятся в различных списках.
13. Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные – в начало этого списка.
14. Ввести строки из файла, записать в список **`ArrayList`**. Выполнить сортировку строк, используя метод **`sort()`** из класса **`Collections`**.
15. Задана строка, состоящая из символов '(', ')', '[', ']', '{', '}'. Проверить правильность расстановки скобок. Использовать стек.
16. Задан файл с текстом на английском языке. Выделить все различные слова. Слова, отличающиеся только регистром букв, считать одинаковыми. Использовать класс **`HashSet`**.
17. Задан файл с текстом на английском языке. Выделить все различные слова. Для каждого слова подсчитать частоту его встречаемости. Слова, отличающиеся регистром букв, считать различными. Использовать класс **`HashMap`**.

Вариант В

1. В кругу стоят N человек, пронумерованных от 1 до N. При ведении счета по кругу вычеркивается каждый второй человек, пока не останется один. Составить две программы, моделирующие процесс. Одна из

программ должна использовать класс **`ArrayList`**, а вторая – **`LinkedList`**. Какая из двух программ работает быстрее? Почему?

2. Задан список целых чисел и число X. Не используя вспомогательных объектов и не изменяя размера списка, переставить элементы списка так, чтобы сначала шли числа, не превосходящие X, а затем числа, большие X.
3. Написать программу, осуществляющую сжатие английского текста. Построить для каждого слова в тексте оптимальный префиксный код по алгоритму Хаффмена. Использовать класс **`PriorityQueue`**.
4. Реализовать класс Graph, представляющий собой неориентированный граф. В конструкторе класса передается количество вершин в графе. Методы должны поддерживать быстрое добавление и удаление ребер.
5. На базе коллекций реализовать структуру хранения чисел с поддержкой следующих операций:
   - добавление/удаление числа;
   - поиск числа, наиболее близкого к заданному (т.е. модуль разницы минимален).
6. Реализовать класс, моделирующий работу N-местной автостоянки. Машина подъезжает к определенному месту и едет вправо, пока не встретится свободное место. Класс должен поддерживать методы, обслуживающие приезд и отъезд машины.
7. Во входном файле хранятся две разреженные матрицы A и B. Построить циклически связанные списки CA и CB, содержащие ненулевые элементы соответственно матриц A и B. Просматривая списки, вычислить: а) сумму S = A + B; б) произведение P = A * B.
8. Во входном файле хранятся наименования некоторых объектов. Построить список C1, элементы которого содержат наименования и шифры данных объектов, причем элементы списка должны быть упорядочены по возрастанию шифров. Затем “сжать” список C1, удаляя дублирующие наименования объектов.
9. Во входном файле расположены два набора положительных чисел; между наборами стоит отрицательное число. Построить два списка C1 и C2, элементы которых содержат соответственно числа 1-го и 2-го набора таким образом, чтобы внутри одного списка числа были упорядочены по возрастанию. Затем объединить списки C1 и C2 в один упорядоченный список, изменяя только значения полей ссылочного типа.
10. Во входном файле хранится информация о системе главных автодорог, связывающих г.Минск с другими городами Беларуси. Используя эту информацию, постройте дерево, отображающее систему дорог республики, а затем, продвигаясь по дереву, определить минимальный по длине путь из г.Минска в другой заданный город. Предусмотреть возможность для последующего сохранения дерева в виртуальной памяти.
11. Один из способов шифрования данных, называемый «двойным шифрованием», заключается в том, что исходные данные при помощи некоторого преобразования последовательно шифруются на некоторые два ключа K1 и K2. Разработать и реализовать эффективный алгоритм, позволяющий находить ключи K1 и K2 по исходной строке и ее зашифрованному варианту. Проверить, оказался ли разработанный способ дей-

ствительно эффективным, протестировав программу для случая, когда оба ключа K1 и K2 являются 20-битными (время ее работы не должно превосходить одной минуты).

12. На плоскости задано N точек. Вывести в файл описания всех прямых, которые проходят более чем через одну точку из заданных. Для каждой прямой указать, через сколько точек она проходит. Использовать класс **`HashMap`**.
13. На клетчатой бумаге нарисован круг. Вывести в файл описания всех клеток, целиком лежащих внутри круга, в порядке возрастания расстояния от клетки до центра круга. Использовать класс **`PriorityQueue`**.
14. На плоскости задано N отрезков. Найти точку пересечения двх отрезков, имеющую минимальную абсциссу. Использовать класс **`TreeMap`**.
15. На клетчатом листе бумаги закрашена часть клеток. Выделить все различные фигуры, которые образовались при этом. Фигурой считается набор закрашенных клеток, достижимых друг из друга при движении в четырёх направлениях. Две фигуры являются различными, если их нельзя совместить поворотом на угол, кратный 90 градусам, и параллельным переносом. Используйте класс **`HashSet`**.
16. Дана матрица из целых чисел. Найти в ней прямоугольную подматрицу, состоящую из максимального количества одинаковых элементов. Использовать класс **`Stack`**.
17. Реализовать структуру "черный ящик", хранящую множество чисел и имеющую внутренний счетчик K, изначально равный нулю. Структура должна поддерживать операции добавления числа в множество и возвращение K-го по минимальности числа из множества.
18. На прямой гоночной трассе стоит N автомобилей, для каждого из которых известны начальное положение и скорость. Определить, сколько произойдет обгонов.
19. На прямой гоночной трассе стоит N автомобилей, для каждого из которых известны начальное положение и скорость. Вывести первые K обгонов.

Тестовые задания к главе 10

Вопрос 10.1.

Какой интерфейс наиболее пригоден для создания класса, содержащего несортированные уникальные объекты?

1) `Set;`
2) `List;`
3) `Map;`
4) `Vector;`
5) нет правильного ответа.

Вопрос 10.2.

Какие из фрагментов кода создадут объект класса **`ArrayList`** и добавят элемент?

1) `ArrayList a = new ArrayList(); a.add("0");`

2) `ArrayList a = new ArrayList(); a[0]="0";`
3) `List a = new List(); a.add("0");`
4) `List a = new ArrayList(10); a.add("0");`

Вопрос 10.3.

Какой интерфейс реализует класс **`Hashtable`**?

1) `Set;`
2) `Vector;`
3) `AbstractMap;`
4) `List;`
5) `Map.`

Вопрос 10.4.

Дан код:

```
import java.util.*;
class Quest4 {
   public static void main (String args[]) {
      Object ob = new HashSet();
      System.out.print((ob instanceof Set) + ", ");
      System.out.print(ob instanceof SortedSet);
}}
```

Что будет выведено при попытке компиляции и запуска программы?

1) true, false;
2) true, true;
3) false, true;
4) false, false;
5) ничего из перечисленного.

Вопрос 10.5.

Какие из приведенных ниже названий являются именами интерфейсов пакета **`java.util`**?

1) `SortedMap;`
2) `HashMap;`
3) `HashSet;`
4) `SortedSet;`
5) `Stack;`
6) `AbstractMap.`