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

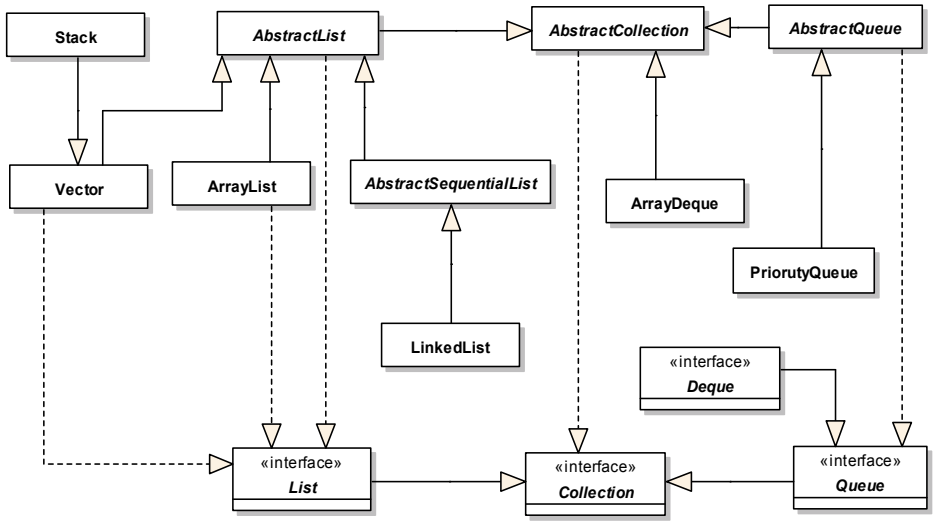

Рис. 10.1. Иерархия наследования списков

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.