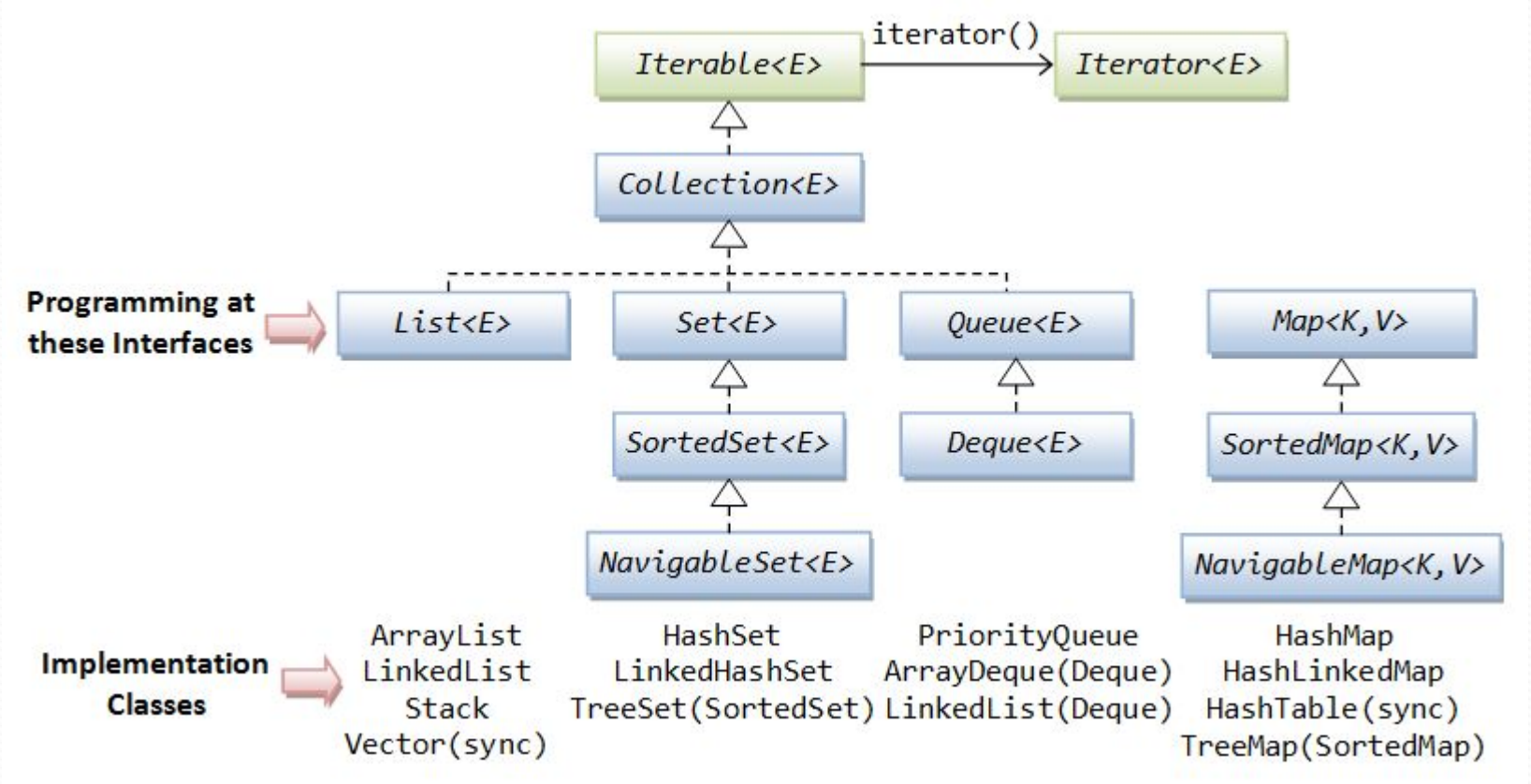
Лекция 8 Коллекции. Часть 1

1. Collections Framework
2. Интерфейс Collection
3. Описание основных структур данных
4. Интерфейс List
5. Классы ArrayList, LinkedList
6. Интерфейс Set и классы HashSet, LinkedHashSet
7. Интерфейс SortedSet и класс TreeSet
8. Сравнение объектов
9. Интерфейс NavigableSet
10. Интерфейс Queue, Deque и классы ArrayDeque, LinkedList, PriorityQueue

Что такое Java Collections Framework?

- **Коллекции** – это хранилища, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним.
- Добавлены в версии J2SE 1.2.
- Collection Framework состоит из 3-х частей:
  - ***Интерфейсы***
  - ***Реализации (классы)***
  - ***Алгоритмы***

Иерархия интерфейсов коллекции



Интерфейс ***Collection***

Интерфейс Collection - вершина иерархии коллекций, который определяет наименьший набор характеристик, реализуемых всеми коллекциями.

Методы интерфейса ***Collection***

boolean add(E obj)

Добавляет *obj* к вызывающей коллекции. Возвращает *true*, если *obj* был добавлен к коллекции.

Методы интерфейса ***Collection***

boolean addAll (Collection<? extends E> c)

Добавляет все элементы к вызывающей коллекции. Возвращает *true*, если операция удалась (то есть все элементы добавлены). В противном случае возвращает *false*.

Методы интерфейса ***Collection***

void clear()

Удаляет все элементы вызывающей коллекции.

Методы интерфейса ***Collection***

boolean contains(Object obj)

Возвращает *true*, если *obj* является элементом вызывающей коллекции. В противном случае возвращает *false*.

Методы интерфейса ***Collection***

boolean containsAll(Collection<?> c)

Возвращает *true*, если вызывающая коллекция содержит все элементы *c*. В противном случае возвращает *false*.

Методы интерфейса ***Collection***

boolean isEmpty()

Возвращает *true*, если вызывающая коллекция пуста. В противном случае возвращает *false*.

Методы интерфейса ***Collection***

boolean remove(Object obj)

Удаляет один экземпляр *obj* из вызывающей коллекции. Возвращает *true*, если элемент удален. В противном случае возвращает *false*.

Методы интерфейса ***Collection***

boolean removeAll(Collection<?> c)

Удаляет все элементы из вызывающей коллекции. Возвращает *true*, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает *false*.

Методы интерфейса ***Collection***

int size()

Возвращает количество элементов, содержащихся в коллекции.

Методы интерфейса *Collection*

```
default boolean removeIf(Predicate<? super E> filter)
```

Удаляет элементы из коллекции, соответствующие заданному условию.

Интерфейс Collection

Метод	Описание
boolean add(E obj)	Добавляет obj к вызывающей коллекции. Возвращает true, если obj был добавлен к коллекции.
boolean addAll (Collection<? extends E> c)	Добавляет все элементы к вызывающей коллекции. Возвращает true, если операция удалась (то есть все элементы добавлены). В противном случае возвращает false.
void clear()	Удаляет все элементы вызывающей коллекции.
boolean contains(Object obj)	Возвращает true, если obj является элементом вызывающей коллекции. В противном случае возвращает false.
boolean containsAll(Collection<?> c)	Возвращает true, если вызывающая коллекция содержит все элементы c. В противном случае возвращает false.

Интерфейс Collection

Метод	Описание
boolean equals(Object obj)	Возвращает true, если вызывающая коллекция и obj эквивалентны. В противном случае возвращает false.
int hashCode()	Возвращает хешкод вызывающей коллекции.
boolean isEmpty()	Возвращает true, если вызывающая коллекция пуста. В противном случае возвращает false.
Iterator<E> iterator()	Возвращает итератор для вызывающей коллекции.
boolean remove(Object obj)	Удаляет один экземпляр obj из вызывающей коллекции. Возвращает true, если элемент удален. В противном случае возвращает false.
boolean removeAll(Collection<?> c)	Удаляет все элементы из вызывающей коллекции. Возвращает true, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает false.

Интерфейс Collection

Метод	Описание
default boolean removeIf(Predicate <? **super** E> filter)	Удаляет элементы из коллекции, соответствующие заданному условию.
boolean retainAll(Collection <?> c)	*Удаляет все элементы кроме входящих из вызывающей коллекции. Возвращает true, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает false.*
int size()	*Возвращает количество элементов, содержащихся в коллекции.*
Object[] toArray()	*Возвращает массив, содержащий все элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции.*

Интерфейс Collection

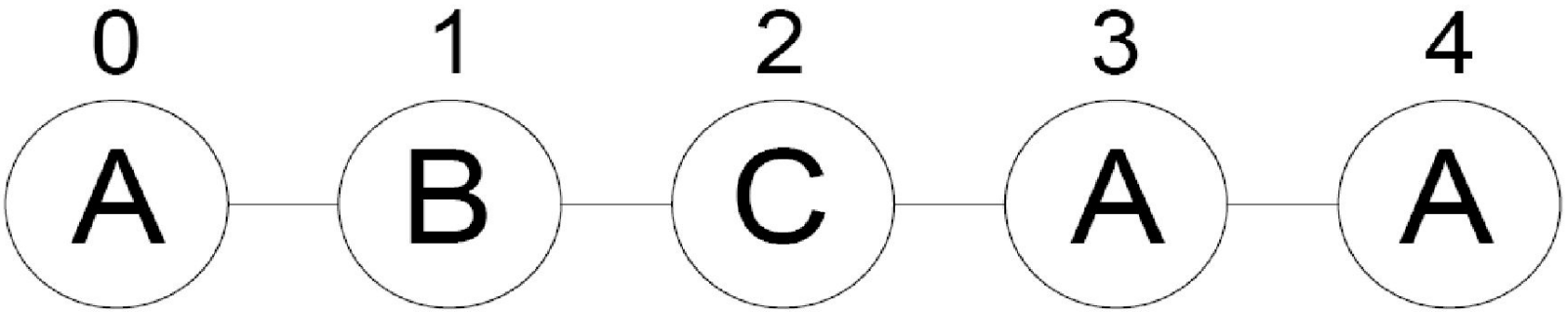
Метод	Описание
<T> T [] toArray (T array[])	Возвращает массив, содержащий элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции. Если размер массива array равен количеству элементов, он возвращается. Если размер array меньше количества элементов, создается и возвращается новый массив нужного размера. Если размер array больше количества элементов, то элементы, следующие за последним из коллекции, устанавливаются равными null. Если любой из элементов коллекции имеет тип, не являющийся подтипом array, возбуждается исключение ArrayStoreException.

Основные структуры данных

- Список
- Стек
- Очередь
- Множество

Список

Список — упорядоченный набор элементов, для каждого из которых хранится указатель на следующий (или для двусвязного списка и на следующий и на предыдущий) элементы списка. Иногда называется sequence. Разрешаются повторы.



Стек

Стек — это коллекция, элементы которой получают по принципу «последний вошел, первый вышел» *(Last-In-First-Out или LIFO)*. Это значит, что мы будем иметь доступ только к последнему добавленному элементу.

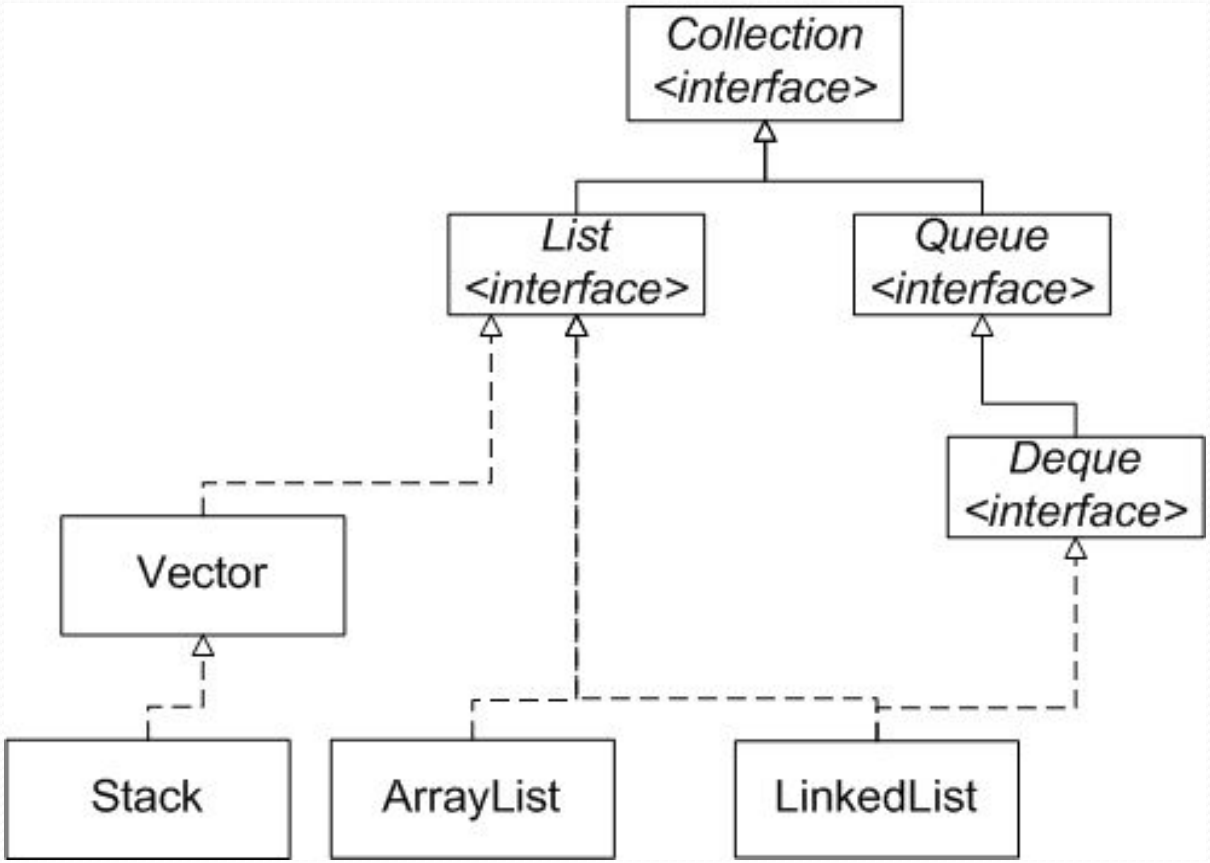
Очередь

Очереди очень похожи на стеки. Они также не дают доступа к произвольному элементу, но, в отличие от стека, элементы помещаются *(enqueue)* и забираются *(dequeue)* с разных концов. Такой метод называется «первый вошел, первый вышел» *(First-In-First-Out или FIFO)*. То есть забирать элементы из очереди мы будем в том же порядке, что и помещали. Как реальная очередь или конвейер.

Множество

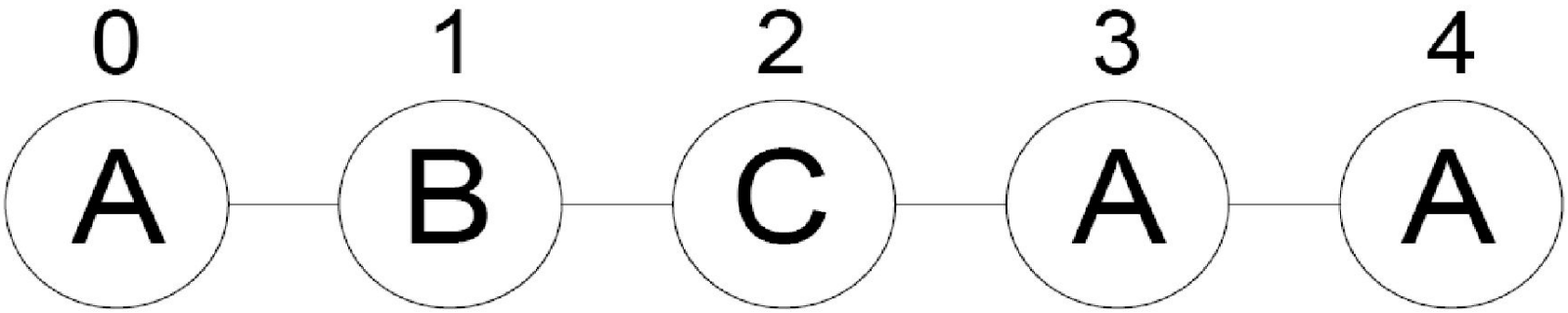
Множество — неупорядоченный набор элементов, без повторов.

Интерфейс ***List***



Интерфейс *List*

Интерфейс ***List*** сохраняет последовательность добавления элементов и позволяет осуществлять доступ к элементу по индексу.



Методы интерфейса ***List***

void add(int index, E obj)

Вставляет *obj* в вызывающий список в позицию, указанную в *index*. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. То есть никакие элементы не перезаписываются.

Методы интерфейса ***List***

boolean addAll (int index, Collection<? extends E> c)

Вставляет все элементы в вызывающий список, начиная с позиции, переданной в *index*. Все ранее существовавшие элементы за точкой вставки смещаются вверх. То есть никакие элементы не перезаписываются. Возвращает *true*, если вызывающий список изменяется, и *false* в противном случае.

Методы интерфейса ***List***

E get (int index)

Возвращает объект, сохраненный в указанной позиции вызывающего списка.

Методы интерфейса ***List***

int indexOf(Object obj)

Возвращает индекс первого экземпляра *obj* в вызывающем списке. Если *obj* не содержится в списке, возвращается -1.

Методы интерфейса *List*

int lastIndexOf(Object obj)

Возвращает индекс последнего экземпляра *obj* в вызывающем списке. Если *obj* не содержится в списке, возвращается -1.

Методы интерфейса ***List***

E remove(int index)

Удаляет элемент из вызывающего списка в позиции *index* и возвращает удаленный элемент. Результирующий список уплотняется, то есть элементы, следующие за удаленным, сдвигаются на одну позицию назад.

Методы интерфейса ***List***

E set (int index, E obj)

Присваивает *obj* элементу, находящемуся в списке в позиции *index*.

Методы интерфейса ***List***

List<E> subList (int start, int end)

Возвращает список, включающий элементы от *start* до *end-1* из вызывающего списка. Элементы из возвращаемого списка также сохраняют ссылки в вызывающем списке.

Интерфейс List

Метод	Описание
void add(int index, E obj)	Вставляет obj в вызывающий список в позицию, указанную в index. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. То есть никакие элементы не перезаписываются.
boolean addAll (int index, Collection<? extends E> c)	Вставляет все элементы в вызывающий список, начиная с позиции, переданной в index. Все ранее существовавшие элементы за точкой вставки смещаются вверх. То есть никакие элементы не перезаписываются. Возвращает true, если вызывающий список изменяется, и false в противном случае.
E get (int index)	Возвращает объект, сохраненный в указанной позиции вызывающего списка.

Интерфейс List

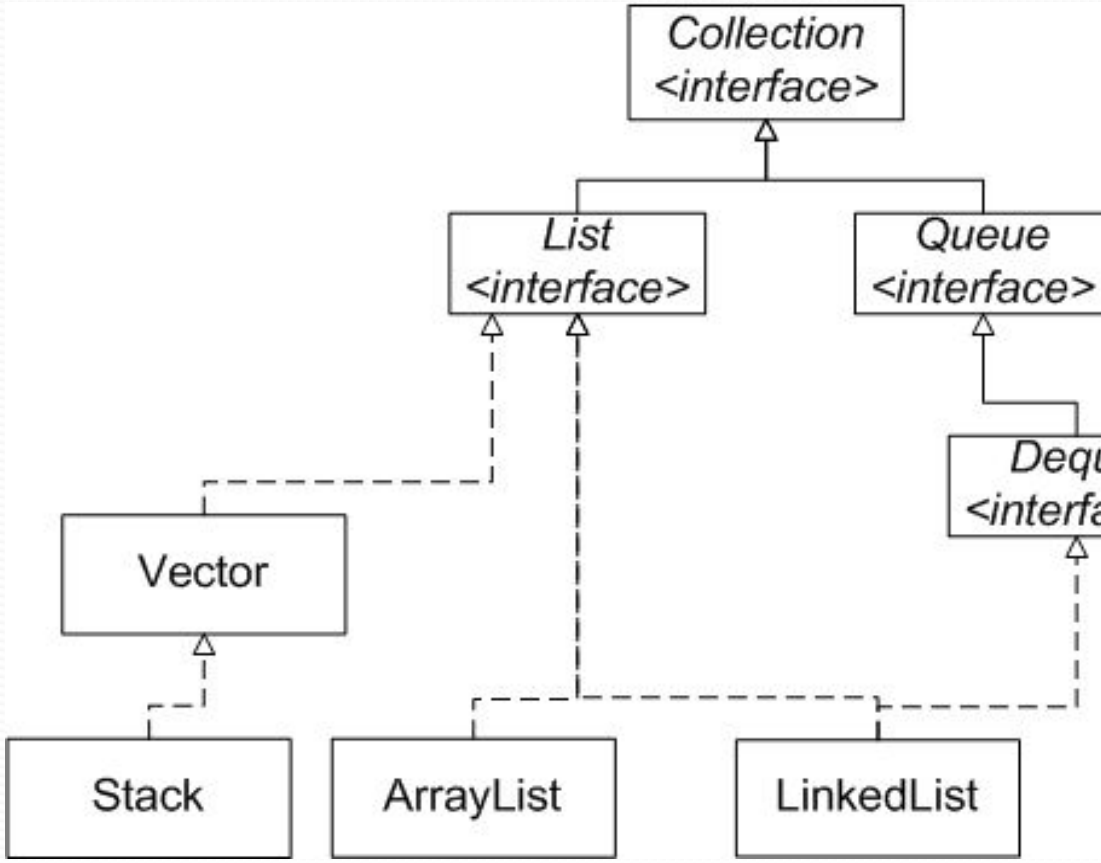
Метод	Описание
int indexOf(Object obj)	Возвращает индекс первого экземпляра obj в вызывающем списке. Если obj не содержится в списке, возвращается -1.
int lastIndexOf(Object obj)	Возвращает индекс последнего экземпляра obj в вызывающем списке. Если obj не содержится в списке, возвращается -1.
ListIterator<E> listIterator() ListIterator<E> listIterator(int index)	Возвращает итератор, указывающий на начало списка. Возвращает итератор, указывающий на заданную позицию в списке.
E remove(int index)	Удаляет элемент из вызывающего списка в позиции index и возвращает удаленный элемент. Результирующий список уплотняется, то есть элементы, следующие за удаленным, сдвигаются на одну позицию назад.

Интерфейс List

Метод	Описание
E set (int index, E obj)	*Присваивает obj элементу, находящемуся в списке в позиции index.*
default void sort(Comparator<? **super** E> c)	*Сортирует список, используя заданный компаратор (добавлен в версии JDK 8).*
List<E> subList (int start, int end)	*Возвращает список, включающий элементы от start до end-1 из вызывающего списка. Элементы из возвращаемого списка также сохраняют ссылки в вызывающем списке.*

Класс *ArrayList*

- ***ArrayList*** поддерживает динамические массивы, которые могут расти по мере необходимости.
- Элементы ***ArrayList*** могут быть абсолютно любых типов в том числе и ***null***.



Внутреннее представление класса ***ArrayList***

- Объект класса ***ArrayList***, содержит свойства ***elementData*** и ***size***.
- Хранилище значений ***elementData*** есть ни что иное как массив определенного типа (указанного в generic).

Внутреннее представление класса ***ArrayList***

- Когда внутренний массив заполняется, ***ArrayList*** создает внутри себя новый массив.
- Его размер = (размер старого массива * 1,5) +1.

- Все данные копируются из старого массива в новый

- Старый массив удаляется сборщиком мусора.

Конструкторы класса *ArrayList*

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

Достоинства и недостатки класса *ArrayList*

Достоинства

- Быстрый доступ по индексу - O(1).
- Быстрая вставка и удаление элементов с конца - O(1).

Недостатки

- Медленная вставка и удаление элементов в середину – O(n).

Методы класса ***ArrayList*** для добавления элементов

1. ***boolean add(E obj)*** - добавляет obj к вызывающей коллекции. Возвращает ***true***, если obj был добавлен к коллекции. (Интерфейс ***Collection***)
2. ***void add(int index, E obj)*** - вставляет obj в вызывающий список в позицию, указанную в index. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. То есть никакие элементы не перезаписываются. (Интерфейс ***List***)
3. ***E set (int index, E obj)*** - присваивает obj элементу, находящемуся в списке в позиции index. (Интерфейс ***List***)
4. ***boolean addAll (Collection<? extends E> c)*** - добавляет все элементы к вызывающей коллекции. Возвращает ***true***, если операция удалась (то есть все элементы добавлены). В противном случае возвращает ***false***. (Интерфейс ***Collection***)

Добавление эл-в в *ArrayList*

```
public class ArrayListAddDemo {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();

        System.out.println("Начальный размер arrayList: "
                + arrayList.size());

        arrayList.add("C");
        arrayList.add("A");
        arrayList.add("E");
        arrayList.add("B");
        arrayList.add("D");
        arrayList.add("F");
        arrayList.add("F");
        arrayList.add(1, "A2");
        arrayList.set(0, "C2");

        System.out.println("Размер arrayList после добавления: "
                + arrayList.size());
        System.out.println("Содержимое arrayList: " + arrayList);
    }
}
```

Методы класса ***ArrayList*** для удаления элементов

1. ***boolean remove(Object obj)*** - удаляет один экземпляр obj из вызывающей коллекции. Возвращает ***true***, если элемент удален. В противном случае возвращает ***false***. (Интерфейс ***Collection***)
2. ***E remove(int index)*** - удаляет элемент из вызывающего списка в позиции index и возвращает удаленный элемент. Результирующий список уплотняется, то есть элементы, следующие за удаленным, сдвигаются на одну позицию назад. (Интерфейс ***List***)
3. ***boolean removeAll(Collection<?> c)*** - удаляет все элементы из вызывающей коллекции. Возвращает ***true***, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает ***false***. (Интерфейс ***Collection***)
4. ***boolean retainAll(Collection<?> c)*** - удаляет все элементы кроме входящих из вызывающей коллекции. Возвращает ***true***, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает ***false***. (Интерфейс ***Collection***)
5. ***void clear()*** - удаляет все элементы вызывающей коллекции. (Интерфейс ***Collection***)

Удаление эл-в из *ArrayList*

```
public class ArrayListRemoveDemo {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();

        arrayList.add("C");
        arrayList.add("A");
        arrayList.add("E");
        arrayList.add("B");
        arrayList.add("D");
        arrayList.add("F");
        arrayList.add("F");
        arrayList.add(1, "A2");
        arrayList.set(0, "C2");

        System.out.println("Содержимое arrayList: " + arrayList);
        System.out.println("Размер arrayList после добавления: "
                + arrayList.size());

        arrayList.remove("F");
        arrayList.remove(2);

        System.out.println("Размер arrayList после удаления: "
                + arrayList.size());
        System.out.println("Содержимое of arrayList: " + arrayList);
    }
}
```

Пример использования *removeAll()* класса *ArrayList*

```
public class ArrayListRemoveAllDemo {
    public static void main(String[] args) {
        List<String> arrayList =  new ArrayList<>();

        arrayList.add("C");
        arrayList.add("A");
        arrayList.add("E");
        arrayList.add("B");
        arrayList.add("D");
        arrayList.add("F");
        arrayList.add("F");
        arrayList.add(1, "A2");
        arrayList.set(0, "C2");

        List<String> removeElements = List.of("C2", "A2", "AA",
"F");
        System.out.println("Содержимое arrayList до removeAll: "
                + arrayList);

        arrayList.removeAll(removeElements);
        System.out.println("Содержимое arrayList после removeAll: "
                + arrayList);
    }
}
```

Пример использования методов *addAll(), clear()* класса *ArrayList*

```
public class ArrayListDemo2 {
    public static void main(String[] args) {
        List<String> arrayList1 =  new ArrayList<>();
        List<String> arrayList2 = List. of("1", "2");

        arrayList1.add( "A");
        arrayList1.add( "B");
        arrayList1.add( "C");
        arrayList1.add( "D");
        arrayList1.add( "E");
        arrayList1.add( "F");
        System. out.println("arrayList1 до добавления " + arrayList1);

        arrayList1.addAll( 3, arrayList2);
        System. out.println("arrayList1 после добавления " +
arrayList1);

        arrayList1.clear();
        System. out.println("arrayList1 после очистки " + arrayList1);
    }
}
```

Пример использования методов *retainAll()* класса *ArrayList*

```
public class ArrayListRetainAllDemo {
    public static void main(String[] args) {
        List<String> arrayList1 =  new ArrayList<>();
        List<String> arrayList2 = List. of("F", "FF",
"E");

        arrayList1.add( "A");
        arrayList1.add( "A");
        arrayList1.add( "B");
        arrayList1.add( "C");
        arrayList1.add( "D");
        arrayList1.add( "E");
        arrayList1.add( "F");
        arrayList1.add( "F");

        arrayList1.retainAll(arrayList2);
        System.out.println(arrayList1);
    }
}
```

Получение массива из коллекции типа ***ArrayList***

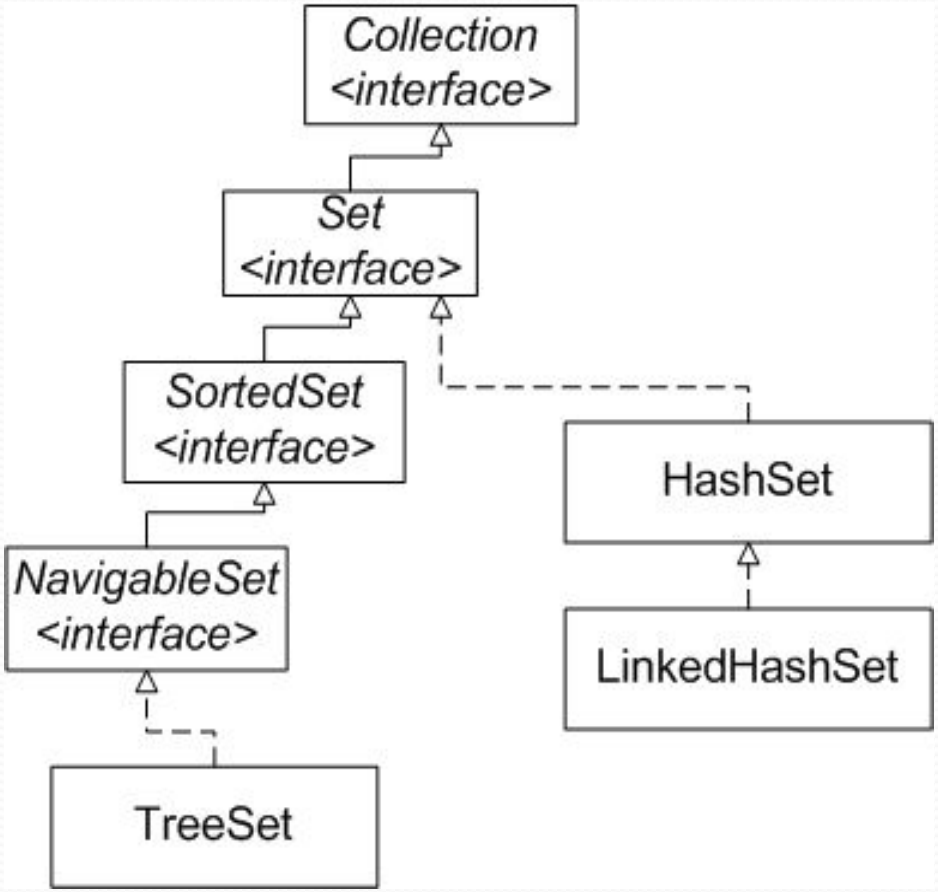
Имеются два варианта метода ***toArray()***, которые объявлены в ***Collection:***

- *Object [] toArray()*
- *<T> T [] toArray(T массив[])*

Получение массива из коллекции типа *ArrayList*

```
public class ArrayListToStringDemo {
    public static void main(String[] args) {
        List<String> arrayList = List.of("C", "A", "E", "B", "D",
"F");
        //1 вариант
        Object[] objectArray = arrayList.toArray();
        System.out.println(Arrays.toString(objectArray));

        //2 вариант
        String[] stringArray1 = new String[arrayList.size()];
        arrayList.toArray(stringArray1);
        System.out.println(Arrays.toString(stringArray1));

        //3 вариант
        String[] stringArray2 = arrayList.toArray( new String[0]);
        System.out.println(Arrays.toString(stringArray2));
    }
}
```

Интерфейс *Set*

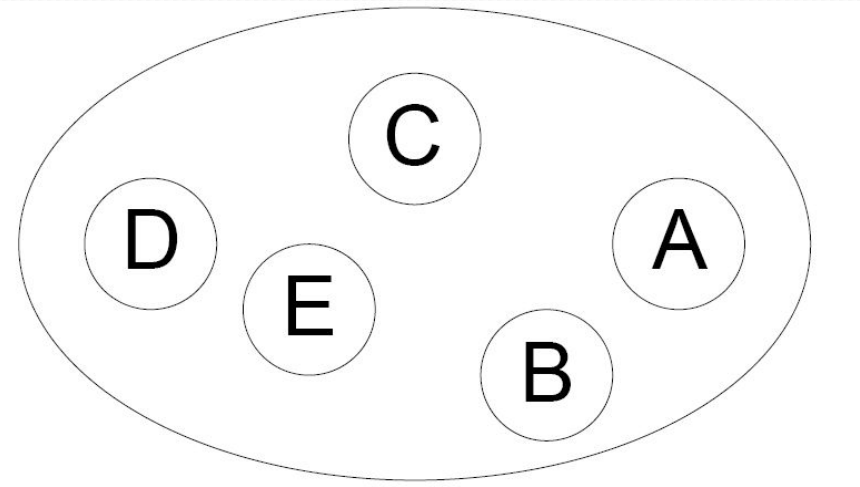


Интерфейс Set

1. Интерфейс ***Set*** определяет множество (набор).
2. Он расширяет ***Collection*** и определяет поведение коллекций, не допускающих дублирования элементов.
3. Таким образом, метод ***add()*** возвращает ***false***, если делается попытка добавить дублированный элемент в набор.
4. Он не определяет никаких собственных дополнительных методов.

Интерфейс Set

6. Интерфейс ***Set*** заботится об уникальности хранимых объектов, уникальность определятся реализацией метода ***equals()*.**



Класс *HashSet*

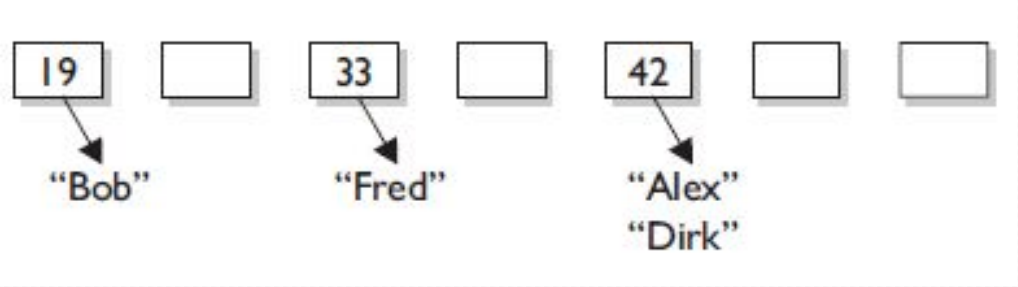
- Класс ***HashSet*** реализует интерфейс ***Set***.
- Он создает коллекцию, которая используется для хранения хеш-таблиц.
- Разрешено добавлять ***null*** значения.

Что такое хэш-таблица?

- Элементы таблицы хранятся в виде пар *ключ-значение*.
- *Ключ* определяет ячейку для хранения *значения*.
- Содержимое ключа служит для определения однозначного значения, называемого *хеш-кодом*.
- Мы можем считать, что ***хеш-код*** это ID объекта, хотя он не должен быть уникальным.
- Этот хеш-код служит далее в качестве индекса, по которому сохраняются данные, связанные с ключом.

Использование hashCode()

Ключ	Алгоритм хеш-кода	Хеш-код
Alex	A(1) + L(12) + E(5) + X(24)	42
Bob	B(2) + O(15) + B(2)	19
Dirk	D(4) + I(9) + R(18) + K(11)	42
Fred	F(6) + R(18) + E(5) + D(4)	33



Правила написания методов ***hashCode()*** и ***equals()***

1. Для одного и того же объекта, хеш-код всегда будет одинаковым.
2. Если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот).
3. Если хеш-коды равны, то входные объекты не всегда равны.
4. Если хеш-коды разные, то и объекты гарантированно разные.

Конструкторы *HashSet*

- **HashSet()** - начальная емкость по умолчанию – 16, коэффициент загрузки – 0,75.
- **HashSet(int initialCapacity)** - Коэффициент загрузки – 0,75.
- **HashSet(int initialCapacity, float loadFactor)**
- **HashSet(Collection C)** – конструктор, добавляющий элементы из другой коллекции.

Начальная емкость и коэффициент загрузки

Начальная емкость (initial capacity)– изначальное количество ячеек (buckets) в хэш-таблице.
Если все ячейки будут заполнены, их количество увеличится автоматически.

Коэффициент загрузки (load factor) – показатель того, насколько заполненным может быть ***HashSet*** до того момента, когда его емкость автоматически увеличится.

Когда количество элементов в ***HashSet*** становится больше, чем ***capacity*loadfactor***, хэш-таблица ре-хэшируется (заново вычисляются хэшкоды элементов, и таблица перестраивается согласно полученным значениям) и количество buckets в ней увеличивается в 2 раза.
Коэффициент загрузки, равный 0,75, в среднем обеспечивает хорошую производительность. Если этот параметр увеличить, тогда уменьшится нагрузка на память (так как это уменьшит количество операций ре-хэширования и перестраивания), но это повлияет на операции добавления и поиска. Чтобы минимизировать время, затрачиваемое на ре-хэширование, нужно правильно подобрать параметр начальной емкости.

Класс *HashSet*

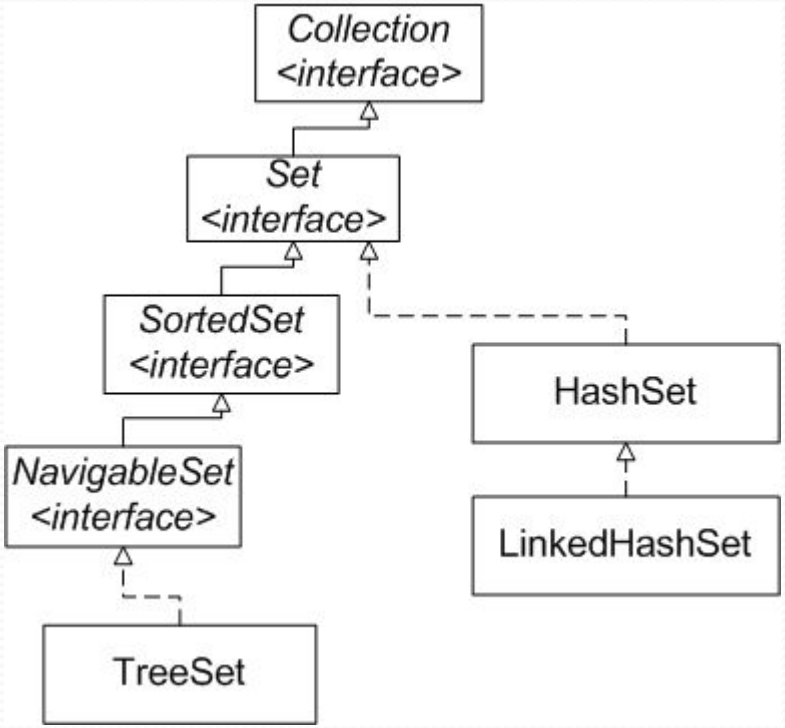
- Выгода от хеширования состоит в том, что оно обеспечивает постоянство время выполнения операций ***add(), contains(), remove() и size(),*** даже для больших наборов – O(1). В худшем случае (если один bucket) время будет O(n) для Java 7 и O(log n) для Java 8.
- Класс ***HashSet*** не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств.

Пример использования класса ***HashSet***

```
public class HashSetDemo {
    public static void main(String[] args) {
        Set<String> hashSet =  new
HashSet<>();

        hashSet.add("Харьков");
        hashSet.add("Киев");
        hashSet.add("Львов");
        hashSet.add("Кременчуг");
        hashSet.add("Харьков");

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

Класс ***LinkedHashSet***

- Класс ***LinkedHashSet*** расширяет ***HashSet***, не добавляя никаких новых методов.
- ***LinkedHashSet*** поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор.

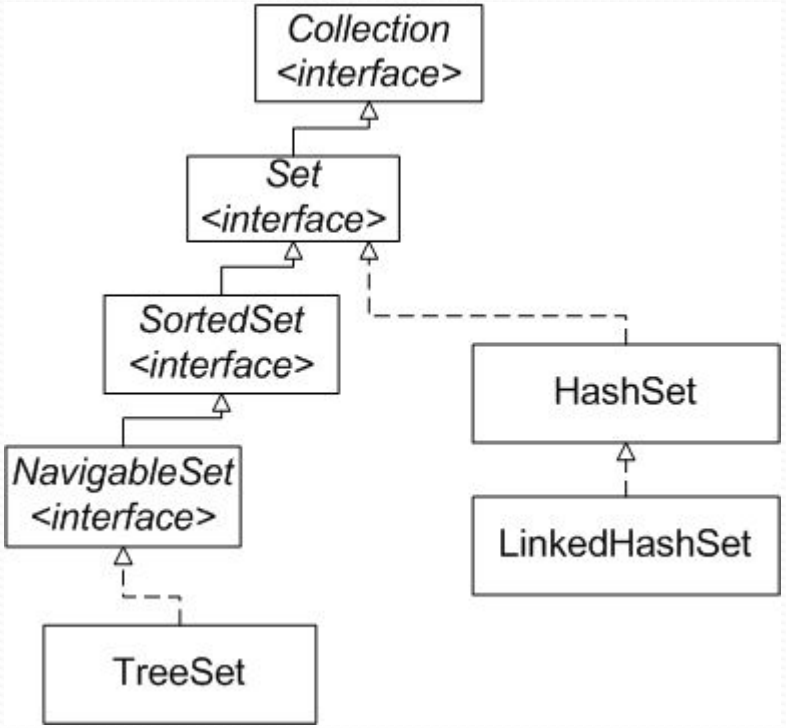


Пример класса *LinkedHashSet*

```
public class LinkedHashSetDemo {
    public static void main(String[] args) {
        Set<String> linkedHashSet = new LinkedHashSet<>();

        linkedHashSet.add("Харьков");
        linkedHashSet.add("Киев");
        linkedHashSet.add("Львов");
        linkedHashSet.add("Кременчуг");
        linkedHashSet.add("Харьков");

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

Интерфейс ***SortedSet***

•Интерфейс ***SortedSet*** из пакета ***java.util***, расширяющий интерфейс ***Set***, описывает упорядоченное множество, отсортированное по естественному порядку возрастания его элементов или по порядку, заданному реализацией интерфейса ***Comparator***.



Класс *TreeSet*

- ***TreeSet<E>*** – реализует интерфейс ***NavigableSet<E>,*** который поддерживает элементы в отсортированном по возрастанию порядке.
- Объекты сохраняются в отсортированном порядке по нарастающей.
- Обработка операций удаления и вставки объектов происходит медленнее O(log (n)), чем в хэш-множествах, но быстрее, чем в списках.

Пример использования класса *TreeSet*

```
public class TreeSetDemo1 {
    public static void main(String[] args) {
        SortedSet<String> treeSet = new TreeSet<>();

        treeSet.add("Харьков");
        treeSet.add("Киев");
        treeSet.add("Львов");
        treeSet.add("Кременчуг");
        treeSet.add("Харьков");

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

Методы интерфейса *SortedSet*

Метод	Описание
Comparator<? super E> comparator ()	*Возвращает компаратор сортированного множества. Если для множества применяется естественный по рядок сортировки, возвращается null.*
E first ()	*Возвращается первый элемент вызывающего сортированного множества.*
SortedSet<E> headSet(E end)	*Возвращается SortedSet, содержащий элементы из вызывающего множества, которые предшествуют end. Элемент из возвращенного множества имеют также ссылки в вызывающем объекте.*
E last ()	*Возвращается последний элемент вызывающего сортированного множества.*
SortedSet<E> subSet (E start, E end)	*Возвращается SortedSet, который включает элементы, находящиеся между start и end-1. Элементы из возвращенного множества имеют также ссылки в вызывающем объекте.*
SortedSet<E> tailSet (E start)	*Возвращается SortedSet, содержащий элементы из вызывающего множества, которые следуют за end. Элементы из возвращенного множества имеют также ссылки в вызывающем объекте.*

Пример использования методов *subSet(), headSet(), tailSet(), first(), last()*

```
public class TreeSetDemo2 {
    public static void main(String[] args) {
        SortedSet<String> treeSet =  new TreeSet<>();

        treeSet.add("Харьков");
        treeSet.add("Киев");
        treeSet.add("Львов");
        treeSet.add("Кременчуг");
        treeSet.add("Харьков");

        System.out.println(treeSet);

        SortedSet<String> subSet = treeSet.subSet( "Киев",
"Львов");
        System.out.println("SubSet: " + subSet);

        System.out.println("HeadSet: " +
treeSet.headSet("Львов"));
        System.out.println("TailSet: " +
treeSet.tailSet("Львов"));
        System.out.println("Первый элемент: " + treeSet.first());
        System.out.println("Последний элемент: " +
treeSet.last());
    }
```

Сравнение объектов

Существует два способа сравнения объектов:

- С помощью интерфейса ***Comparable<E>***.
- С помощью интерфейса ***Comparator<E>*.**

Интерфейс ***Comparable<T>***

- Для того, чтобы объекты можно было сравнить и сортировать, они должны реализовать интерфейс ***Comparable<T>***.
- Интерфейс ***Comparable<T>*** содержит один единственный метод ***int compareTo(T item),*** который сравнивает текущий объект с объектом, переданным в качестве параметра.
- Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр.
- Если метод вернет положительное число, то, наоборот, после второго объекта.
- Если метод возвращает ноль, значит, оба объекта равны.

Пример использования *Comparable<T>*

```
public class Person implements Comparable<Person> {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName,  int age)
{
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    …

    @Override
    public int compareTo(Person anotherPerson) {
        int anotherPersonAge = anotherPerson.getAge();
        return this.age - anotherPersonAge;
    }
}
```

Пример использования *Comparable<T>*

```
public class ComparePersonDemo {
    public static void main(String[] args) {
        SortedSet<Person> set = new TreeSet<>();
        set.add(new Person("Саша", "Иванов", 36));
        set.add(new Person("Маша", "Петрова", 23));
        set.add(new Person("Даша", "Сидорова", 34));
        set.add(new Person("Вася", "Иванов", 25));
        set.forEach(System.out::println);
    }
}
```

Интерфейс ***Comparator<T>***

- Интерфейс ***Comparator<T>*** содержит метод:
  int *compare(T o1, T o2).*
- Метод ***compare*** также возвращает числовое значение - если оно отрицательное, то объект *o1* предшествует объекту *o2*, иначе - наоборот. А если метод возвращает ноль, то объекты равны.
- Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс.

Пример использования *Comparator<T>*

```
public class PersonComparator implements Comparator<Person> {
     @Override
     public int compare(Person o1, Person o2) {
         return o1.getLastName().compareTo(o2.getLastName());
     }
}
```

Пример использования *Comparator<T>*

```
public class PersonComparatorDemo {
    public static void main(String[] args) {
        PersonComparator personComparator =new PersonComparator();
        SortedSet<Person> set =new TreeSet<>(personComparator);
        set.add(new Person("Саша", "Иванов", 36));
        set.add(new Person("Маша", "Петрова", 23));
        set.add(new Person("Даша", "Сидорова", 34));
        set.add(new Person("Вася", "Иванов", 25));
        //Обратите внимание - было добавлено 4 элемента, но распечатано 3
        set.forEach(System.out::println);
    }
}
```

Пример использования *Comparator.comparing()*

```
public class PersonComparingDemo {
    public static void main(String[] args) {
        Comparator<Person> personComparator =
                Comparator.comparing(Person::getLastName)
                        .thenComparing(Person::getAge);
        SortedSet<Person> set = new TreeSet<>(personComparator);
        set.add(new Person("Саша", "Иванов", 36));
        set.add(new Person("Маша", "Петрова", 23));
        set.add(new Person("Даша", "Сидорова", 34));
        set.add(new Person("Вася", "Иванов", 25));
        set.forEach(System.out::println);
    }
}
```

Конструкторы класса *TreeSet*

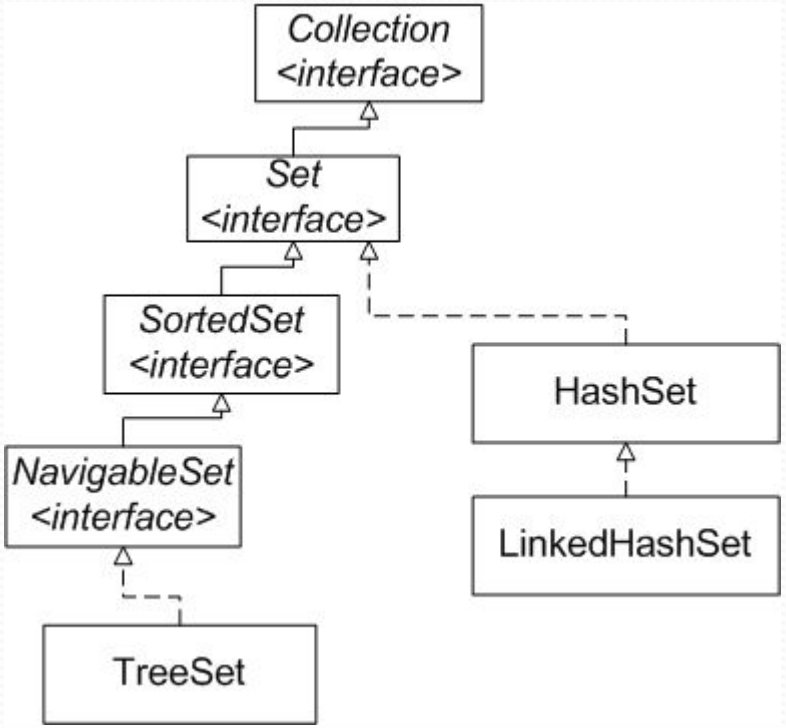
TreeSet имеет следующие конструкторы:

- ***TreeSet()***
- ***TreeSet(Collection<? extends E> collection)***
- ***TreeSet(Comparator<? super E> comparator)***
- ***TreeSet(SortedSet<E> sortedSet)***

- https://javarush.ru/quests/lectures/questcollections.level06.lecture07

Интерфейс ***NavigableSet***

- Интерфейс ***NavigableSet*** появился в Java SE 6.
- Он расширяет ***SortedSet*** и добавляет методы для более удобного поиска по коллекции.



Интерфейс ***NavigableSet***

Метод	Описание
E ***ceiling****(E obj)*	Ищет в наборе наименьший элемент e, для которого истинно e >= obj. Если такой элемент найден, он возвращается. В противном случае возвращается ***null***.
E ***floor****(E obj)*	Ищет в наборе наибольший элемент e, для которого истинно e <= obj. Если такой элемент найден, он возвращается. В противном случае возвращается ***null***.
E ***higher****(E obj)*	Ищет в наборе наибольший элемент e, для которого истинно e > obj. Если такой элемент найден, он возвращается. В противном случае возвращается ***null***.
E ***lower****(E obj)*	Ищет в наборе наименьший элемент e, для которого истинно e < obj. Если такой элемент найден, он возвращается. В противном случае возвращается *null*.

Интерфейс ***NavigableSet***

Метод	Описание
NavigableSet<E> ***headSet****(* *E upperBound, boolean incl)*	Возвращает ***NavigableSet***, включающий все элементы вызывающего набора, меньшие upperBound. Результирующий набор поддерживается вызывающим набором.
NavigableSet<E> ***subSet****(* *E lowerBound, boolean lowIncl,* *E upperBound, boolean highIncl)*	Возвращает ***NavigableSet***, включающий все элементы вызывающего набора, которые больше lowerBound и меньше upperBound. Если lowIncl равно true, то элемент, равный lowerBound, включается. Если highIncl равно true, также включается элемент, равный upperBound.

Интерфейс ***NavigableSet***

Метод	Описание
E ***pollLast****()*	Возвращает последний элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наибольшим значением. Возвращает ***null*** в случае пустого набора.
E ***pollFirst****()*	Возвращает первый элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наименьшим значением. Возвращает ***null*** в случае пустого набора.
Iterator<E> ***descendingIterator****()*	Возвращает итератор, перемещающийся от большего к меньшему, другими словами, обратный итератор.
NavigableSet<E> ***descendingSet()***	Возвращает ***NavigableSet***, представляющий собой обратную версию вызывающего набора. Результирующий набор поддерживается вызывающим набором.

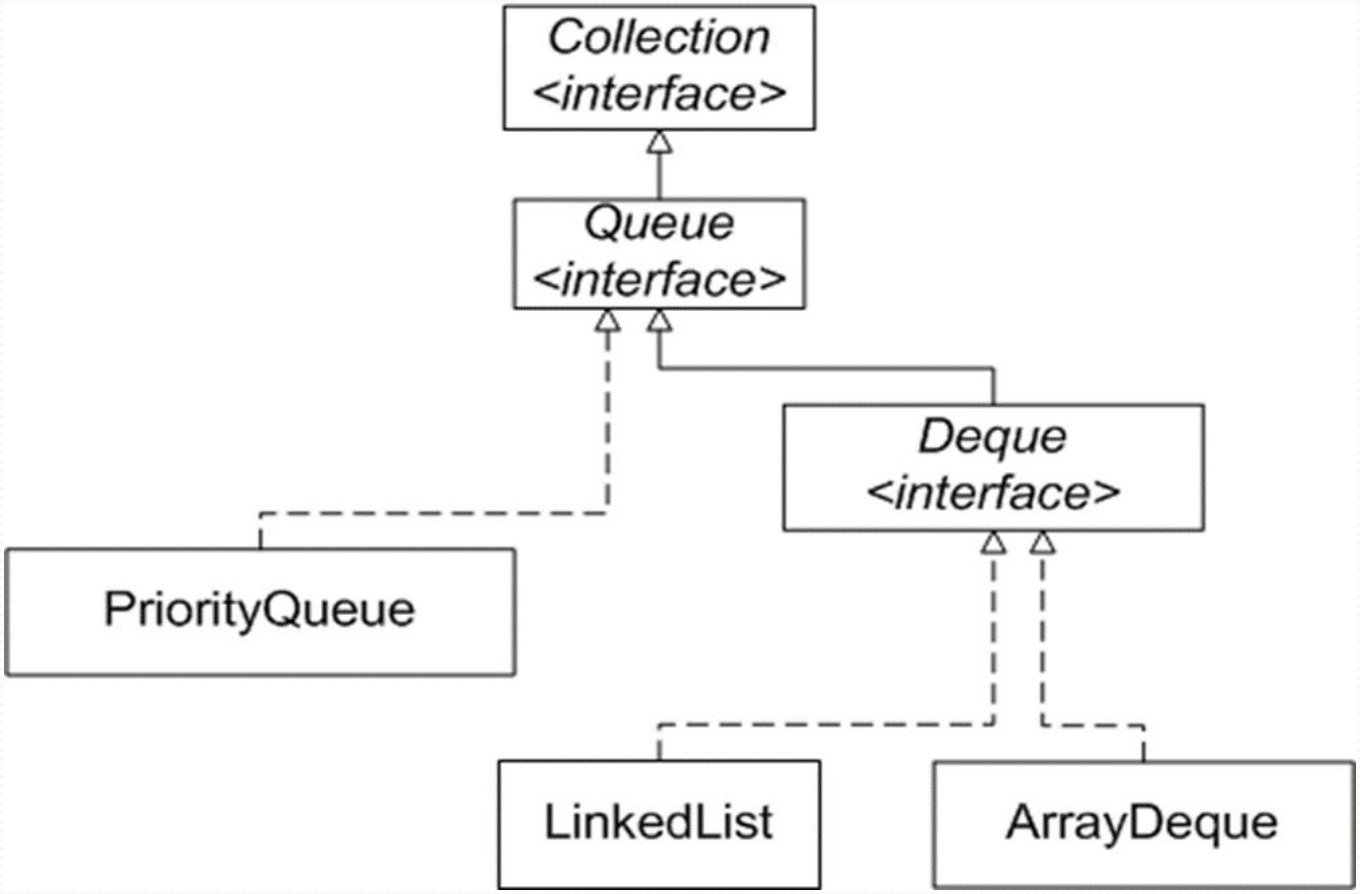
Пример использования *NavigableSet*

```
public class Ferry {
    public static void main(String[] args) {
        NavigableSet<Integer> times =  new TreeSet<>();
        times.add(1205);
        times.add(1505);
        times.add(1545);
        times.add(1830);
        times.add(2010);
        times.add(2100);
        // Java 5 версия
        System.out.println("Java 5");
        SortedSet<Integer> subset = times.headSet( 1600);
        System.out.println("Последний паром перед 16:00 - " + subset.last());

        SortedSet<Integer> tailSet = times.tailSet( 2000);
        System.out.println("Первый паром после 20:00 - " + tailSet.first());
        System.out.println();

        // Java 6 версия исполтзует новые методы lower() и higher()
        System.out.println("Java 6");
        System.out.println("Последний паром перед 16:00 - " +
times.lower(1600));
        System.out.println("Первый паром после 20:00 - " + times.higher(2000));
    }
```

Интерфейс ***Queue***

- Интерфейс ***Queue*** расширяет ***Collection*** и объявляет поведение очередей, которые представляют собой список с дисциплиной "первый вошел первый вышел“ (***FIFO***).
- Существуют разные типы очередей, в которых порядок основан на некотором критерии.
- Очереди не могут хранить ***null***.

Интерфейс ***Queue***



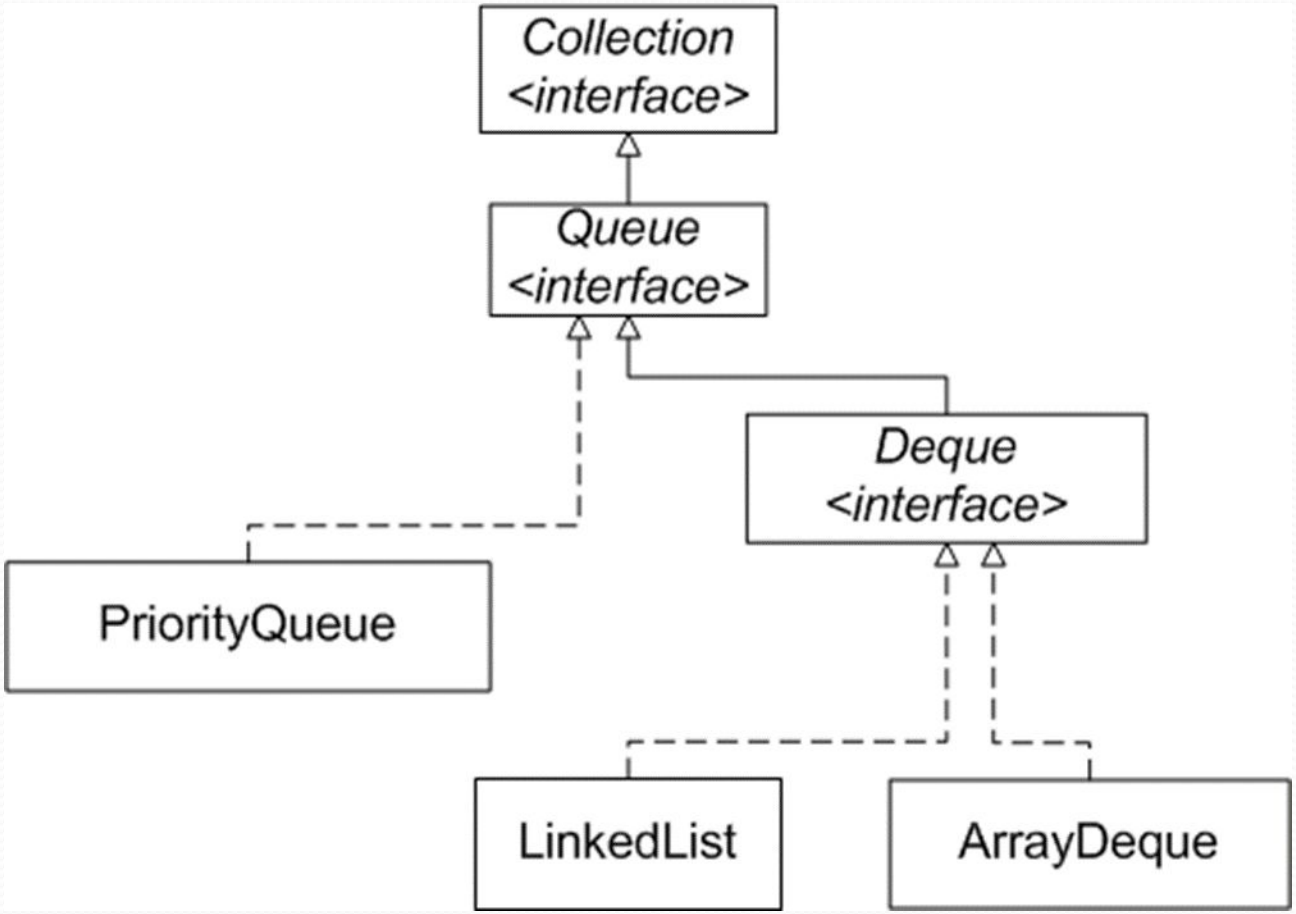
Методы *Queue*

Метод	*Описание*
E element()	*Возвращает элемент из головы очереди. Элемент не удаляется. Если очередь пуста, инициируется исключение* ***NoSuchElementException****.*
E remove()	*Удаляет элемент из головы очереди, возвращая его. Инициирует исключение* ***NoSuchElementException****, если очередь пуста.*
E peek()	*Возвращает элемент из головы очереди. Возвращает null, если очередь пуста. Элемент не удаляется.*
E poll()	*Возвращает элемент из головы очереди и удаляет его. Возвращает null, если очередь пуста.*
boolean offer(E obj)	*Пытается добавить obj в очередь. Возвращает* ***true****, если obj добавлен, и* ***false*** *в противном случае.* Отличие от метода *add()* - метод *add()* выбросит исключение ***IllegalStateException*** если элемент не может быть добавлен в очередь.

Пример методов *offer(), peek(), poll()*

```
public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.offer("Харьков");
        queue.offer("Киев");
        queue.offer("Кременчуг");
        queue.offer("Львов");

        System.out.println(queue.peek());

        String town;
        while ((town = queue.poll()) != null) {
            System.out.println(town);
        }
    }
}
```

Интерфейс ***Deque***



Интерфейс ***Deque***

- Интерфейс ***Deque (double-ended queue)*** появился в Java 6.
- Он расширяет ***Queue*** и описывает поведение двунаправленной очереди.
- Двунаправленная очередь может функционировать как стандартная очередь FIFO либо как стек LIFO.

Методы *Deque*

Метод	*Описание*
void addFirst(E obj)	*Добавляет obj в голову двунаправленной очереди. Возбуждает исключение IllegalStateException, если в очереди ограниченной емкости нет места.*
void addLast(E obj)	*Добавляет obj в хвост двунаправленной очереди. Возбуждает исключение IllegalStateException, если в очереди ограниченной емкости нет места.*
E getFirst()	*Возвращает первый элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключение NoSuchElementException.*
E getLast()	*Возвращает последний элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключения NoSuchElementException.*

Методы *Deque*

Метод	*Описание*
boolean offerFirst (E obj)	*Пытается добавить obj в голову двунаправленной очереди. Возвращает true, если obj добавлен, и false в противном случае. Таким образом, этот метод возвращает false при попытке добавить obj в полную двунаправленную очередь ограниченной емкости.*
boolean offerLast(E obj)	*Пытается добавить obj в хвост двунаправленной очереди. Возвращает true, если obj добавлен, и false в против ном случае.*
E pop()	*Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение NoSuchElementException, если очередь пуста.*
void push(E obj)	*Добавляет элемент в голову двунаправленной очереди. Если в очереди фиксированного объема нет места, возбуждает исключение IllegalStateException.*

Методы *Deque*

Метод	*Описание*
E peekFirst()	*Возвращает элемент, находящийся в голове двунаправленной очереди. Возвращает null, если очередь пуста. Объект из очереди не удаляется.*
E peekLast()	*Возвращает элемент, находящийся в хвосте двунаправленной очереди. Возвращает null, если очередь пуста. Объект из очереди не удаляется.*
E pollFirst()	*Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возвращает null, если очередь пуста.*
E pollLast()	*Возвращает элемент, находящийся в хвосте двунаправленной очереди, одновременно удаляя его из очереди. Возвращает null, если очередь пуста.*

Методы *Deque*

Метод	*Описание*
E removeLast()	*Возвращает элемент, находящийся в конце двунаправленной очереди, удаляя его в процессе. Возбуждает исключение NoSuchElementException, если очередь пуста.*
E removeFirst()	*Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение NoSuchElementException, если очередь пуста.*
boolean removeLastOccurrence(Object obj)	*Удаляет последнее вхождение obj из двунаправленной очереди. Возвращает true в случае успеха и false если очередь не содержала obj.*
boolean removeFirstOccurrence(Object obj)	*Удаляет первое вхождение obj из двунаправленной очереди. Возвращает true в случае успеха и false, если очередь не содержала obj.*

Класс *ArrayDeque*

- ***ArrayDeque*** создает двунаправленную очередь.
- Конструкторы:

  ArrayDeque(); // создает пустую двунаправленную очередь с вместимостью 16 элементов

  ArrayDeque(Collection<? extends E> c); // создает двунаправленную очередь из элементов коллекции с в том порядке, в котором они возвращаются итератором коллекции с.

  ArrayDeque(int numElements); // создает пустую двунаправленную очередь с вместимостью numElements.

Пример использования класса *ArrayDeque*

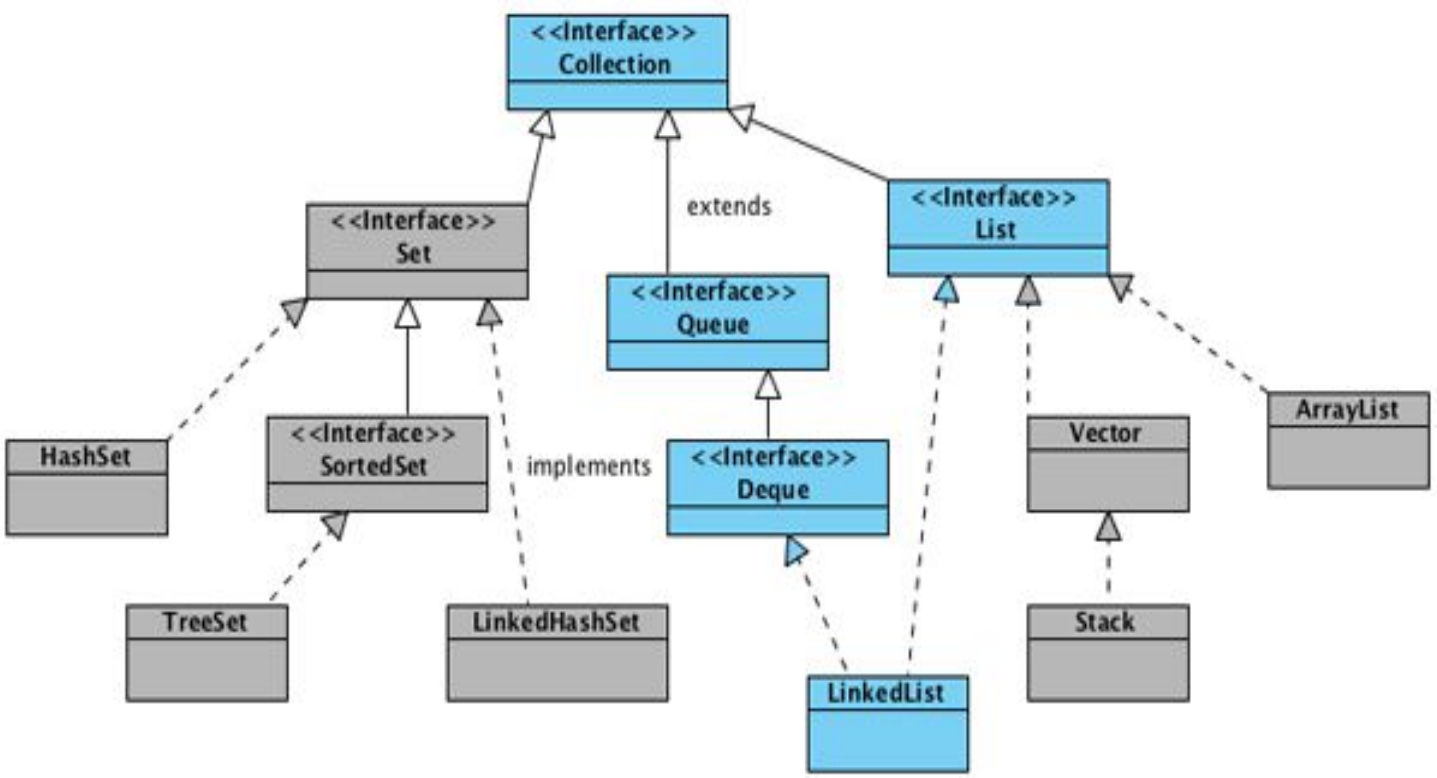
```
public class ArrayDequeExample {
    public static void main(String[] args) {
        Deque<String> stack = new ArrayDeque<>();
        Deque<String> queue = new ArrayDeque<>(2);
        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
        System.out.println();

        queue.offer("A");
        queue.offer("B");
        queue.offer("C");
        queue.offer("D");
        while (!queue.isEmpty()) {
            System.out.print(queue.remove() + "
");
        }
    }
}
```

Класс *LinkedList*

- Класс ***LinkedList*** реализует интерфейсы ***List, Deque***.
- ***LinkedList*** – это двухсвязный список.

- Конструкторы:
  LinkedList()
  LinkedList(Collection<? extends E> c)

Класс ***LinkedList***



Внутренняя организация класса ***LinkedList***

Внутри класса ***LinkedList*** существует *static inner* класс ***Entry***, с помощью которого создаются новые элементы.

```
private static class Entry<E>
{
    E element;
    Entry<E> next;
    Entry<E> prev;

    Entry(E element, Entry<E> next, Entry<E> prev)
    {
        this.element = element;
        this.next = next;
        this.prev = prev;
    }
}
```

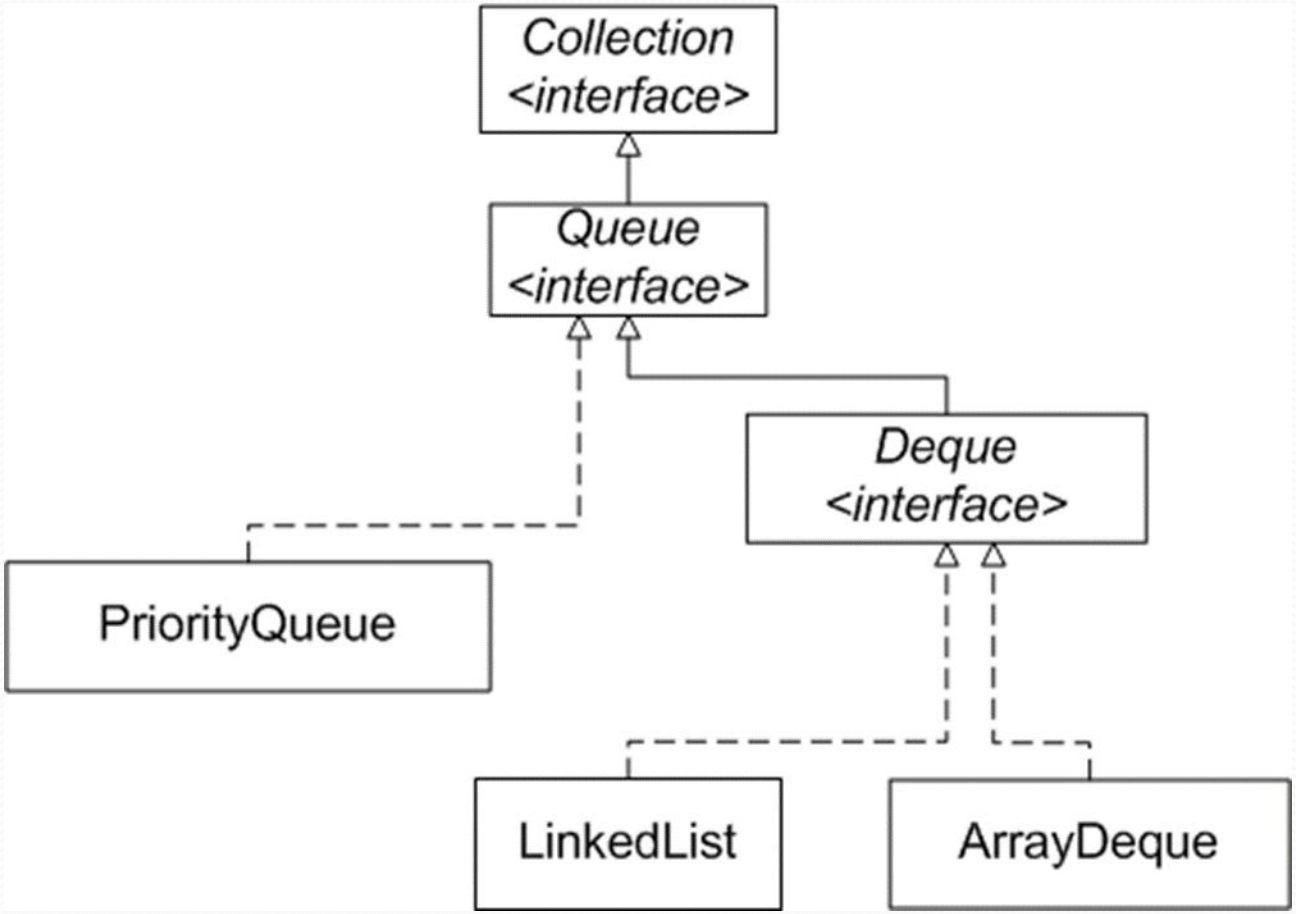
Пример использования *LinkedList*

```
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList<String> list =  new LinkedList<>();

        list.add("F");
        list.add("B");
        list.add("D");
        list.add("E");
        list.add("C");
        list.addLast("Z");
        list.addFirst("A");
        list.add(1, "A2");
        System.out.println("Содержимое: " + list);

        list.remove("F");
        list.remove(2);
        list.removeFirst();
        list.removeLast();
        System.out.println("Содержимое после удаления: " + list);

        String val = list.get(2);
        list.set(2, val + "+");
        System.out.println("Содержимое после изменения: " + list);
    }
```

Класс ***LinkedList***

- Из ***LinkedList*** можно организовать стэк, очередь, или двойную очередь.
- На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n).
- Однако, на добавление и удаление из середины списка, используя ***ListIterator.add()*** и ***ListIterator.remove(),*** потребуется O(1);
- Позволяет добавлять любые значения в том числе и ***null***.

Класс ***PriorityQueue***



Класс *PriorityQueue*

- ***PriorityQueue*** – это класс очереди с приоритетами.
- По умолчанию очередь с приоритетами размещает элементы согласно естественному порядку сортировки используя ***Comparable***.
- Элементу с наименьшим значением присваивается наибольший приоритет.
- Если несколько элементов имеют одинаковый наивысший элемент – связь определяется произвольно.
- Также можно указать специальный порядок размещения, используя ***Comparator.***

Конструкторы *PriorityQueue*

- ***PriorityQueue();*** // создает очередь с приоритетами начальной емкостью 11, размещающую элементы согласно естественному порядку сортировки (Comparable).
- ***PriorityQueue(Collection<? extends E> c);***
- ***PriorityQueue(int initialCapacity);***
- ***PriorityQueue(int initialCapacity, Comparator<? super E> comparator);***
- ***PriorityQueue(PriorityQueue<? extends E> c);***
- ***PriorityQueue(SortedSet<? extends E> c);***

Пример использования *PriorityQueue*

```
public class PriorityQueueExample {
    public static void main(String[] args) {
        Queue<String> queue1 = new PriorityQueue<>();
        queue1.offer("Киев");
        queue1.offer("Харьков");
        queue1.offer("Львов");
        queue1.offer("Кременчуг");
        queue1.offer("Кременчуг");
        System.out.print("Priority queue с Comparable: ");
        while (queue1.size() > 0) {
            System.out.print(queue1.remove() + " ");
        }
        System.out.println();

        PriorityQueue<String> queue2
                = new PriorityQueue<>(5, Collections.reverseOrder());
        queue2.offer("Киев");
        queue2.offer("Харьков");
        queue2.offer("Львов");
        queue2.offer("Кременчуг");
        queue2.offer("Кременчуг");
        System.out.print("Priority queue с Comparator: ");
        while (queue2.size() > 0) {
            System.out.print(queue2.remove() + " ");
        }
    }
}
```