Мультимедийный курс

Программирование на Java

Лекция 7
Collections Framework - фреймверк коллекций объектов

Автор:
Борисенко В.П.

Лаборатория Интернет-технологий

Часть 1

Коллекции

Контейнеры (коллекции)

- В пакет **java.util** входит одна из самых эффективных подсистем java - каркас коллекций Collections Framework
- Каркас коллекций - это сложная иерархия интерфейсов и классов, реализующих современную технологию управления наборами (группами, коллекциями, контейнерами) объектов
- **Коллекциями** называют структуры, предназначенные для хранения *однотипных данных ссылочного типа*
- Все коллекции Java предназначены для хранения объектов, т.е. потомков класса **Object**

Контейнеры (коллекции)

- **Типизированные (параметризованные) коллекции ,** которые появились в Java 5, позволяют ограничить попадание объектов несоответствующего типа в коллекцию на этапе компиляции

Контейнеры (коллекции)

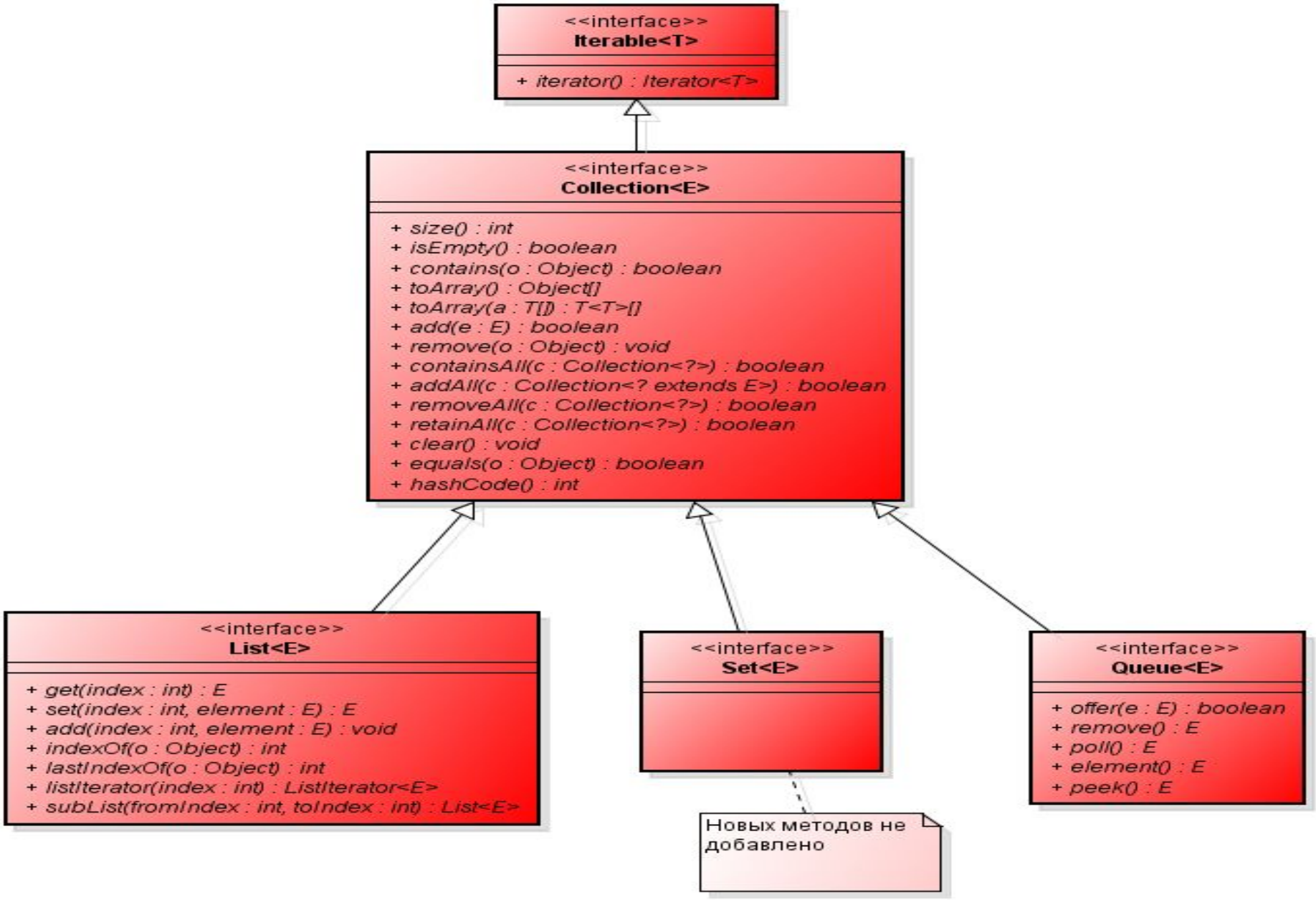
- На вершине библиотеки контейнеров Java расположены **два основных интерфейса**, которые представляют два принципиально разных вида коллекций:
  - интерфейс **Collection<E>** – **группы отдельных объектов** , сформированная по определенным правилам (вершина иерархии остальных коллекций)
  - интерфейс **Map<K,V>** карта отображения– **набор пар объектов «ключ - значение»**, с возможностью выборки по ключу

Массивы vs. Коллекции

- И массивы, и коллекции являются объектами
- Массивы **не могут изменять размер**
- Коллекции **не могут оперировать с примитивными типами**
  - При передаче в коллекцию примитивных типов они автоматически преобразовываются в объекты с помощью **процедуры Autoinboxing**

```
Map<Integer, Integer> map = new
                                HashMap<Integer, Integer>();
map.put(5, 42);
System.out.println(map.get(new Integer(5)));
```

Интерфейс Collection



Интерфейс Collection

- **Collection<E>** представляет собой группу объектов
- Правила хранения элементов задаются нижележащими интерфейсами, сам же интерфейс **Collection<E>** в java.util **прямых реализаций не имеет**

- Интерфейс **Collection** расширяется тремя способами:
  - **интерфейс List<E>** – упорядоченный список, который, хранит элементы в порядке вставки;
  - **интерфейс Set<E>** – множество, в котором нельзя хранить повторяющиеся элементы
  - **Интерфейс Queue<E>** - очередь которая реализует FIFO–буфер

Часть 2

Списки

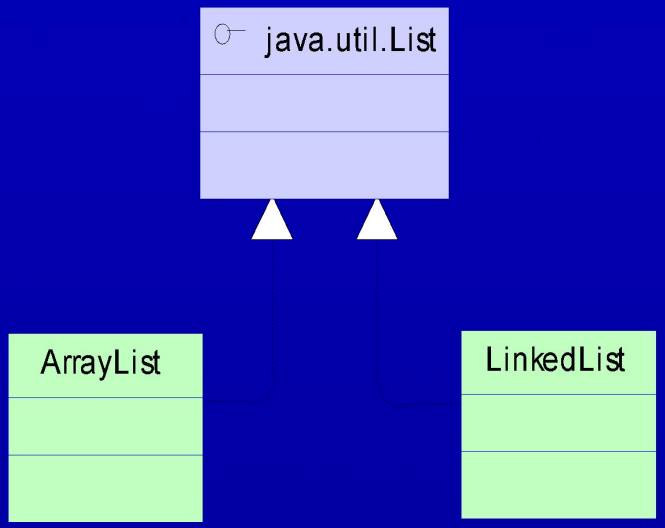
Иерархия наследования списков

Интерфейс List

- **List<E>** – это список объектов

- Объекты хранятся **в порядке их добавления** в список

- В пакете **java.util** имеется 2 основных класса, реализующих интерфейс List:
  - **ArrayList<E>** – в нем для хранения элементов используется *массив*
  - **LinkedList <E>**– для хранения элементов используется *двусвязный список*



Класс ArrayList

- Класс **ArrayList <E>** представляет **собой список динамической длины**
- **Данные внутри класса хранятся во внутреннем массиве**
- Удаление и добавление элементов для такой коллекции представляет собой ресурсоемкую задачу, поэтому объект **ArrayList<E> лучше всего подходит для хранения неизменяемых списков**

Класс ArrayList

- По умолчанию при создании нового объекта **ArrayList** создается **внутренний массив длиной 10 элементов**

```
Collection<Integer>  cl = new
ArrayList<Integer> ();
```

- Можно также создать **ArrayList**, **задав его начальную длину**

```
Collection<Integer>  cl = new
ArrayList<Integer> (100);
```

- Если длины внутреннего массива не хватает для добавления нового объекта, внутри класса **создается новый массив большего объема, и все элементы старого массива копируются в новый**

Класс LinkedList

- Класс **LinkedList <E>** реализует базовый интерфейс **List <E>** и представляет собой список динамической длины. Данные внутри него хранятся в виде связного списка

- 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**

- **LinkedList<E>** выполняет операции вставки и удаления в середине списка более эффективно чем **ArrayList**

Класс LinkedList

- У **LinkedList<E>** представлен ряд методов, не входящих в интерфейс **List**:
  - **addFirst**() и **addLast**() - добавить в начало и в конец списка
  - **removeFirst**() и **removeLast**() - удалить первый и последний элементы
  - **getFirst**() и **getLast**() - получить первый и последний элементы

Интерфейс Queue<E>

- Класс LinkedList<E> реализует интерфейс **Queue<E>**, т. е. такому списку легко придать свойства **очереди**

- **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

Интерфейс Deque <E>

Интерфейс Deque определяет **«двунаправленную» очередь** и, соответственно, методы доступа к первому и последнему элементам двусторонней очереди

Методы обеспечивают **удаление, вставку и обработку элементов**

Интерфейс Deque <E>

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

Интерфейс Deque <E>

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<E>

Доступ к элементам списков

- Доступ к элементам списка возможен
  - по индексу
  - с помощью итератора (Iterator)
    - С явным объявлением итератора
    - В цикле foreach

- Доступ по индексу

```
for (int i = 0; i < list.size(); i++){
    MyClass elem = (MyClass) list.get(i); // Коллекция не
                                          //   типизированная
    elem.doSome();
}
```

  - Для навигации по **LinkedList<E>** при большом количестве объектов использование доступа по индексу ***неэффективно***

Доступ к элементам списков

- Доступ с помощью цикла foreach

```
List<String> list = new ArrayList<String>();
// Вывод list
for (String str : list) {
    System.out.println(str);
}
```

Итераторы (Iterator<E>)

- **Итератор** – это вспомогательный объект, используемый для перемещения в одном направлении по коллекции объектов. Он позволяет написать универсальный код, который не зависит от типа контейнера

- Работа с итераторами производится через интерфейс Iterator<E>, который специфицирует методы:
  - boolean **hasNext**() – проверяет есть ли еще элементы в коллекции
  - Object **next**() – выдает очередной элемент коллекции
  - void **remove**() – удаляет последний выбранный элемент из коллекции.

Итераторы (Iterator)

- Получить итератор для прохода коллекции можно с помощью метода iterator(), который определен у интерфейса Collection**<E>**

```
for (Iterator<String> iter = collection.iterator(); iter.hasNext();)
 {
     MyClass element = (MyClass) iter.next();
     element.doSome();
 }
```

- В случае, если в процессе навигации по коллекции ее содержимое изменилось (например, из другого потока), методы доступа к элементам коллекции по итератору будут бросать исключение ConcurrentModificationException

ListIterator

- **ListIterator<E> более мощная** разновидность Iterator<E>, поддерживаемая только классами List

- ListIterator<E> **является двусторонним**, он может выдавать индексы и значения следующего и предыдущего элемента

- Для создания ListIterator<E> изначально установленного на элемент с индексом n используется вызов ListIterator(n)

ArrayList: index vs. Iterator

```
ArrayList<Integer>  list = new ArrayList();
for (int i=0; i<100000; i++)
  list.add(i);
long a = System.currentTimeMillis();
for (int i=0, n=list.size(); i < n; i++)
    list.get(i);
System.out.println(System.currentTimeMillis()-a);
a = System.currentTimeMillis();
for (Iterator i=list.iterator(); i.hasNext(); )
    i.next();
System.out.println(System.currentTimeMillis()-a);
```

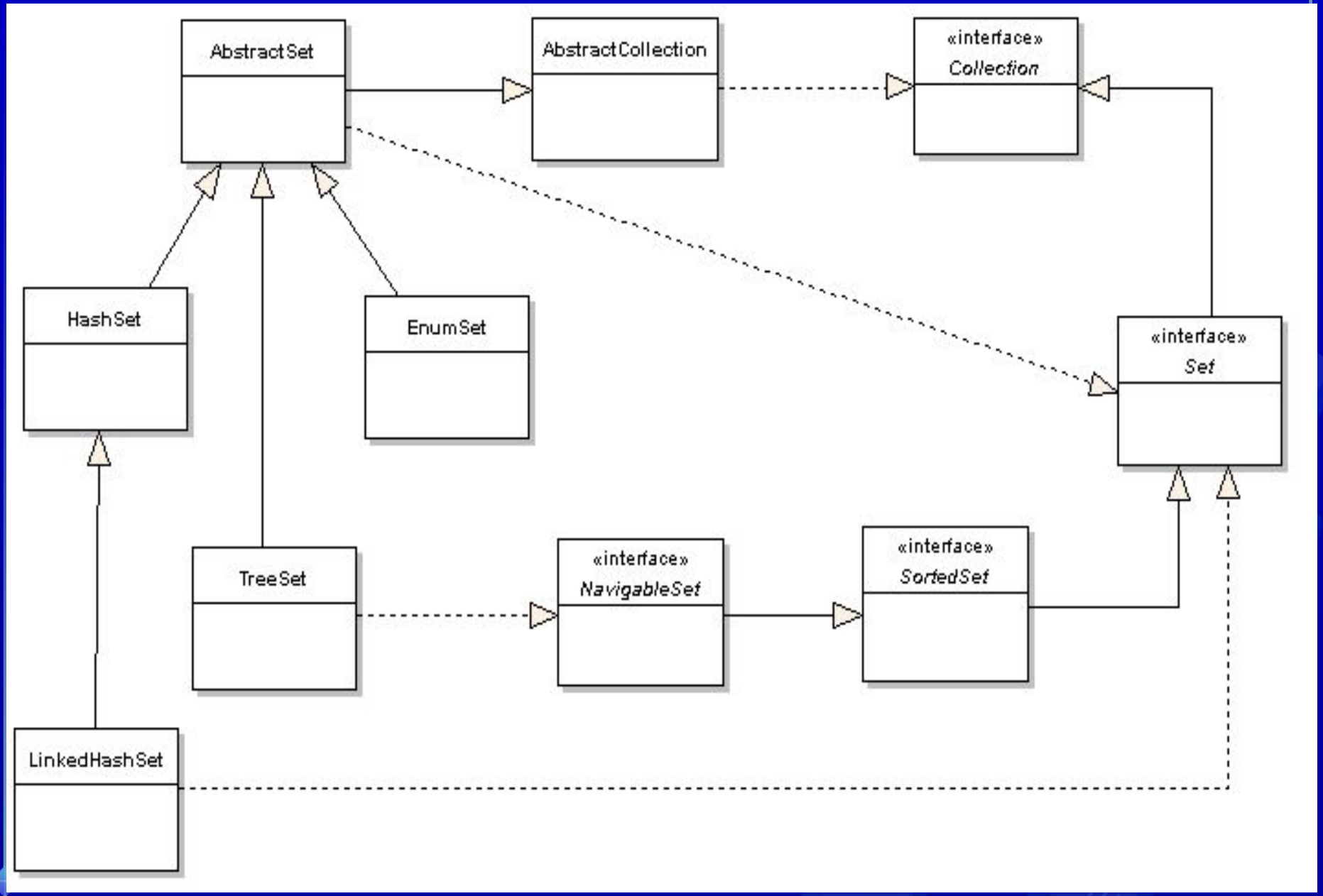
LinkedList: index vs. Iterator

```
LinkedList<Integer> list2 = new LinkedList();
for (int i=0; i<100000; i++)
  list2.add(i);
a = System.currentTimeMillis();
for (int i=0, n=list2.size(); i < n; i++)
    list2.get(i);
System.out.println(System.currentTimeMillis()-a);
a = System.currentTimeMillis();
for (Iterator i=list2.iterator(); i.hasNext(); )
    i.next();
System.out.println(System.currentTimeMillis()-a);
```

ArrayList vs. LinkedList

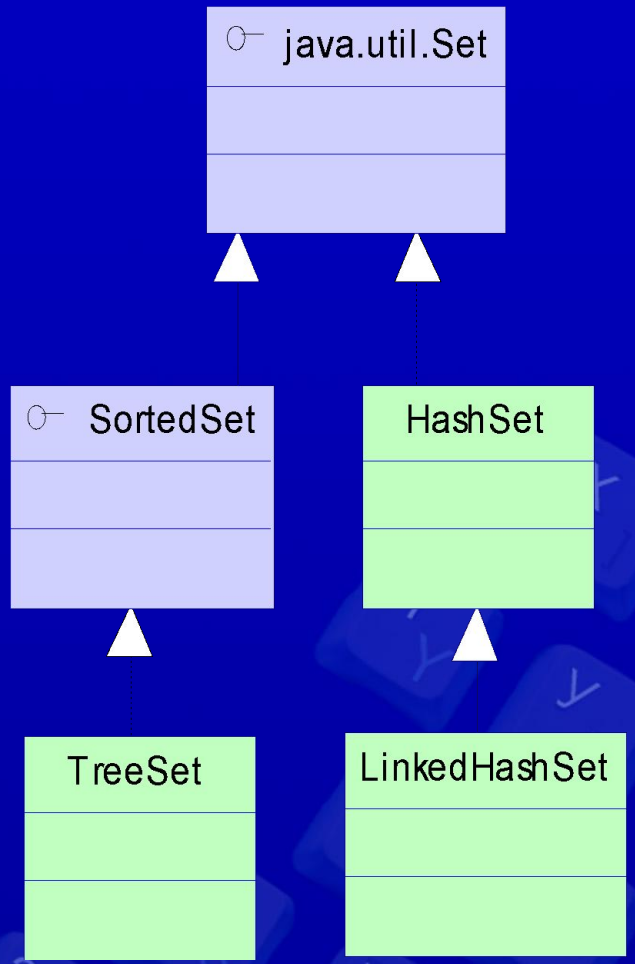
к-во	ArrayList (index)	ArrayList (Iterator)	LinkedList (index)	LinkedList (Iterator)
1000	60	161	10	0
10.000	60	160	250	10
100.000	60	150	353678	30
1.000.000	70	170	...	250

Иерархия наследования множеств



Интерфейс Set

- **Set** <T>– множество неповторяющихся объектов
- Добавление повторяющихся элементов в **Set**<T> не вызывает исключений, но они не попадают в множество
- Для прохода по множеству используется интерфейс итератор



Классы HashSet и LinkedHashSet

- Классы **HashSet<T>** и **LinkedHashSet<T>** реализуют интерфейс **Set<T>**
- Уникальность объектов в них обеспечивается благодаря использованию механизма хеширования
- 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

Классы HashSet и LinkedHashSet

- В **HashSet<T>** объекты хранятся в произвольном порядке
- **LinkedHashSet<T>** является наследником класса HashSet. Он хранит объекты в порядке их добавления

Упорядоченные множества (SortedSet)

- Интерфейс **SortedSet<T>** служит для спецификации упорядоченных множеств
- В JDK его реализация представлена в классе **TreeSet** <T> (для хранения объектов использует бинарное дерево)

Упорядоченные множества (SortedSet)

- 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

Упорядоченные множества (SortedSet)

- При добавлении нового объекта он становится на свое место по порядку в множестве:

```
Set <Integer> sorted = new TreeSet();
sorted.add(new Integer(2));
sorted.add(new Integer(3));
sorted.add(new Integer(1));
System.out.println(sorted); // Распечатает [1, 2, 3]
```

Интерфейс Comparable

- В Java задача задания функции сравнения решается с использованием интерфейсов **Comparable<T>** и **Comparator<T>**
- Интерфейс **Comparable<T>** предназначен для определения так называемого естественного порядка (**natural ordering**)
- Данный интерфейс содержит всего один метод:

```
public int compareTo(Object o) // сравнивает
                               // объект с
                               // другим объектом
```

Интерфейс Comparable

- Метод **compareTo(T t)** возвращает:
  - отрицательное число, если **this < other**;
  - ноль, если **this == other**;
  - положительное число, если **this > other**.
- Дополнительным условием является то, что метод **compareTo**(other) должен возвращать 0 тогда и только тогда, когда метод **equals**(other) возвращает true.

Интерфейс Comparator

- Интерфейс **Comparator** используется, когда метод **compareTo**() уже переопределен, но необходимо задать еще какой-то прядок сортировки

- Интерфейс Comparator содержит один метод:

```
public interface Comparator<E> {
    int compare(T a, T b);
}
```

Интерфейс Comparator

- В этом случае создается отдельный вспомогательный класс, реализующий интерфейс **Comparator<E>**, и уже на основании объекта этого класса будет производиться сортировка

- В этом классе нужно реализовать метод **compare**(T a , T b)

Пример работы с Deque

```
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);
    }
}
```

Пример работы с интерфейсом Deque

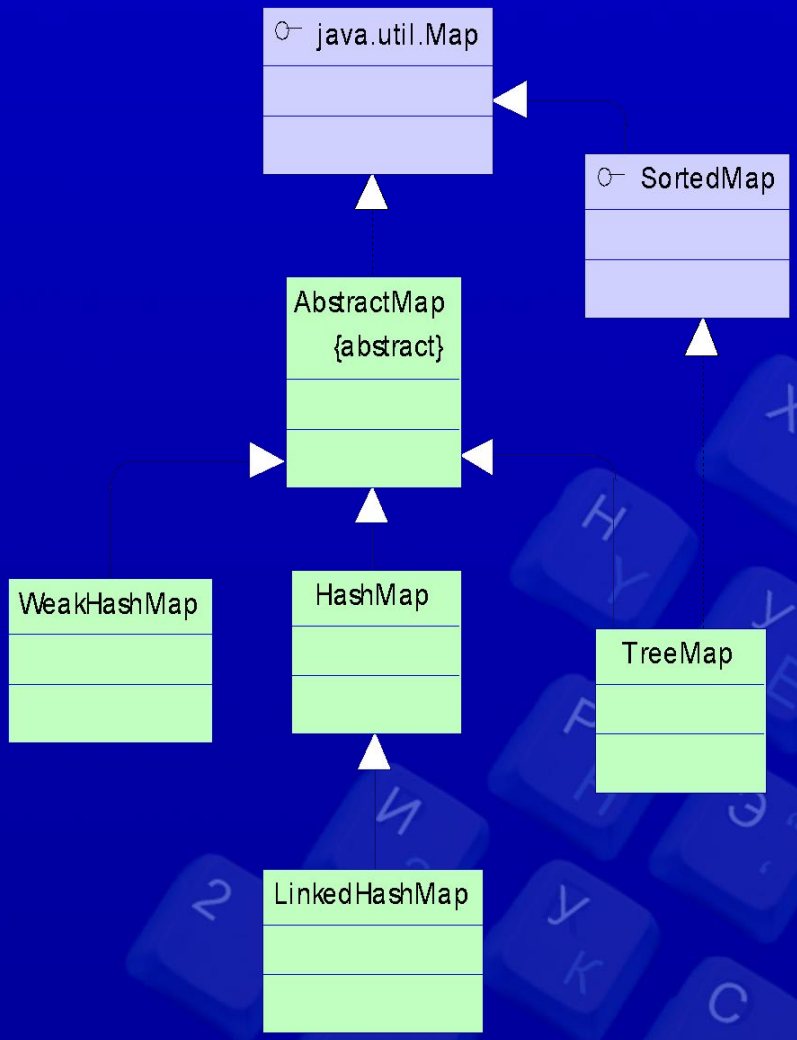
В данном примере реализована работа с интерфейсом Deque. Методы addFirst(), addLast() вставляют элементы в начало и в конец очереди соответственно. Метод add() унаследован от интерфейса Queue и абсолютно аналогичен методу addLast() интерфейса Deque

В результате работы программы на консоль будет выведено:

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

Интерфейс Map

- Интерфейс **Map<K,V>;** часто называют ассоциативным массивом
- **Map<K,V>;** осуществляет отображение (mapping) множества ключей на множество значений. Т.е. объекты хранятся в нем в виде пар <ключ, значение>
- Map<K,V>; позволяет получить значение по ключу.
- В Map<K,V>; не может быть 2-х пар с одинаковым ключом



Методы Map<K,V>

- public void **put**(Object key, Object value) - добавляет новую пару <ключ, значение>
- public Object **get**(Object key) – возвращает value по заданному ключу, или **null**, если ничего не найдено
- public Set **keySet**() – возвращает множество ключей
- boolean **containsKey**(Object key) – возвращает true, если Map содержит пару с заданным ключем

Классы HashMap и LinkedHashMap

- **HashMap<K,V> – расширяет AbstractMap<K,V>, используя хэш-таблицу, в которой ключи отсортированы относительно значений их хэш-кодов**
- **HashMap<K,V>** формирует неупорядоченное множество ключей, т.е. ключи хранятся в произвольном порядке
- **LinkedHashMap<K,V>** содержит ключи в порядке их добавления

Пример с использованием HashMap

```
Map<String, String> map = new HashMap <String, String>();

 // Заполнить его чем-нибудь
 map.put("one", "111");
 map.put("two", "222");
 map.put("three", "333");
 map.put("four", "333");

// Получить и вывести все ключи
System.out.println("Set of keys: " + map.keySet());

 // Получить и вывести значение по ключу
 String val = map.get("one");
 System.out.println("one=" + val);

 // Получить и вывести все значения
System.out.println("Collection of values: " + map.values());

 // Получить и вывести все пары
System.out.println("Set of entries: " + map.entrySet());
```

Внутренний интерфейс Map.Entry

- Интерфейс **Map.Entry<K,V>** позволяет работать с объектом, который представляет собой пару <ключ, значение>
- Каждый элемент ассоциативного массива, описываемого интерфейсом Map**, имеет интерфейсный тип Map.Entry**
- **Метод entrySet(),** определенный в интерфейсе Map, позволят получить все элементы ассоциативного массива в виде множества объектов типа Map.Entry

Внутренний интерфейс Map.Entry

- Интерфейс содержит такие методы как:
  - boolean **equals**(Object o) - проверяет эквивалентность двух пар
  - Object **getKey**() – возвращает ключ элемента (пары.)
  - Object **getValue**() – возвращает значение элемента (пары).
  - Object **setValue**(Object value) –меняет значение элемента (пары)

- Проход по всем Entry :

```
Map<String, Integer> map = new LinkedHashMap<String, Integer>();
map.put("one", 1);
map.put("two", 2);
// …
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println( entry.getKey() + "=" + entry.getValue());
}
```

Синхронизированные коллекции

- В CollectionsFramework большинство коллекций не синхронизировано
  - Кроме устаревших типа Vector
- Чтобы сделать синхронизированную коллекцию, нужно воспользоваться методами класса Collections
  - List synchronizedList(List list)
  - Map synchronizedMap(Map m)
  - Set synchronizedSet(Set s)
  - и т.д.
- В этих методах создается надстройка над передаваемым объектом, реализующая соотв. интерфейс и выполняющая синхронизацию в каждом из методов