Лекция 9 Коллекции. Часть 2

1. Коллекции и обобщения
2. Интерфейсы Iterator, ListIterator
3. Интерфейсы Map, SortedMap, NavigableMap, Map.Entry
4. Классы HashMap, TreeMap, LinkedHashMap,
5. Backed Collections
6. Алгоритмы коллекций
7. Унаследованные коллекции

Коллекции и обобщения <? extends>,<?>

```
public class GenericDemo1 {
    public static void main(String[] args) {
        List<Integer> iList = new ArrayList<>();
        iList.add(1);
        iList.add(5);
        iList.add(8);
        iList.add(9);

        System.out.println(getAverage(iList));

        List<Double> iDouble = new ArrayList<>();
        iDouble.add(1.7);
        iDouble.add(5.7);
        iDouble.add(8.3);
        iDouble.add(9.2);
        //Ошибка компиляции -  List<Integer> iList=iDouble;
        // System.out.println(getAverage(iDouble));
    }

    public static Number getAverage(List<? extends Number> list) {
        double result = 0;
        Number average;
        for (Number d : list) {
            result += d.doubleValue();
        }
        average = result / list.size();
        //  list.add(average);
        return average;
    }
}
```

Коллекции и обобщения <? super>

```
public class GenericDemo2 {
    public static void main(String[] args) {
        List<Box6> boxes = new ArrayList<>();
        List<HeavyBox> heavyBoxes = new ArrayList<>();
        addBox(boxes);
        addBox(heavyBoxes);

        System.out.println(boxes);
        System.out.println(heavyBoxes);
    }

    public static void addBox(List<? super HeavyBox> list) {
        list.add(new HeavyBox());
    }

   /* public static <T> T getBox(List<? super T> list) {
        return list.get(0);
    }*/
}
```

Принцип PECS

- ***PECS*** – Producer Extends Consumer Super.

Перебор содержимого коллекции

Перебор содержимого коллекции может быть осуществлен двумя способами:

- С помощью цикла ***for each.***
- С помощью итератора.

Интерфейс ***Iterator***

- ***Iterator*** позволяет осуществлять обход коллекции и при желании удалять избранные элементы.
- Интерфейс ***Iterator<E>*** используется для доступа к элементам коллекции.
- ***Iterator<E> iterator()*** – возвращает итератор.
- Используйте ***Iterator*** вместо ***for each*** если вам необходимо удалить текущий элемент.

Методы интерфейса *Iterator*

Метод	Описание
boolean hasNext()	Возвращает ***true***, если есть еще элементы. В противном случае возвращает ***false***.
E next()	Возвращает следующий элемент. Если следующий элемент коллекции отсутствует, то метод ***next()*** генерирует исключение ***NoSuchElementException***.
void remove()	Удаляет текущий элемент. Возбуждает исключение ***IllegalStateException***, если предпринимается попытка вызвать ***remove()***, которой не предшествовал вызов ***next()***.
default void forEachRemaining(Consumer<? **super** E> action) {	Выполняет заданное действие для всех оставшихся элементов коллекции.

Пример использования интерфейса *Iterator*

```
public class IteratorDemo {
    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");

        Iterator<String> iterator = arrayList.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            System.out.print(element + " ");
        }
    }
}
```

Перебор содержимого коллекции с помощью *for each*

- Все классы в каркасе коллекций усовершенствованы таким образом, чтобы реализовывать интерфейс ***Iterable***.
- Это означает, что содержимое коллекции можно перебрать, организовав цикл ***for*** в стиле ***for each***.
- Конструкция ***for each*** скрывает итератор, поэтому нельзя вызвать метод ***remove()***.

Интерфейс *ListIterator*

- ***ListIterator*** расширяет интерфейс ***Iterator*** для двустороннего обхода списка и видоизменения его элементов.
- ***ListIterator*** можно получить вызывая метод ***listIterator()*** для коллекций, реализующих ***List***.

Методы интерфейса *ListIterator*

Метод	*Описание*
void add(E obj)	Вставляет ***obj*** перед элементом, который должен быть возвращен следующим вызовом ***next().***
boolean hasNext()	Возвращает ***true***, если есть следующий элемент. В противном случае возвращает ***false***.
boolean hasPrevious()	Возвращает ***true***, если есть предыдущий элемент. В противном случае возвращает ***false***.
E next()	Возвращает следующий элемент. Если следующего нет, инициируется исключение ***NoSuchElementException***.
int nextIndex()	Возвращает индекс следующего элемента. Если следующего нет, возвращается размер списка.
E previous()	Возвращает предыдущий элемент. Если предыдущего нет, инициируется исключение ***NoSuchElementException***.
int previousIndex()	Возвращает индекс предыдущего элемента. Если предыдущего нет, возвращается -1.
void remove()	Удаляет текущий элемент из списка. Если ***remove()*** вызван до ***next()*** или ***previous()***, инициируется исключение ***IllegalStateException***.
void set(E obj)	Присваивает ***obj*** текущему элементу. Это элемент, возвращенный последним вызовом ***next()*** или ***previous()***.

Пример использования интерфейса *ListIterator*

```
public class ListIteratorDemo {
    public static void main(String[] args) {
        List<String> arrayList = Arrays.asList("A", "B", "C", "D");

        ListIterator<String> listIterator = arrayList.listIterator();
        while (listIterator.hasNext()) {
            String element = listIterator.next();
            listIterator.set(element + "+");
        }

        System.out.print("Измененный arrayList в обратном порядке: ");
        while (listIterator.hasPrevious()) {
            String element = listIterator.previous();
            System.out.print(element + " ");
        }
    }
}
```

Виды коллекций

- Отсортированные
- Не отсортированные
- Упорядоченные
- Неупорядоченные

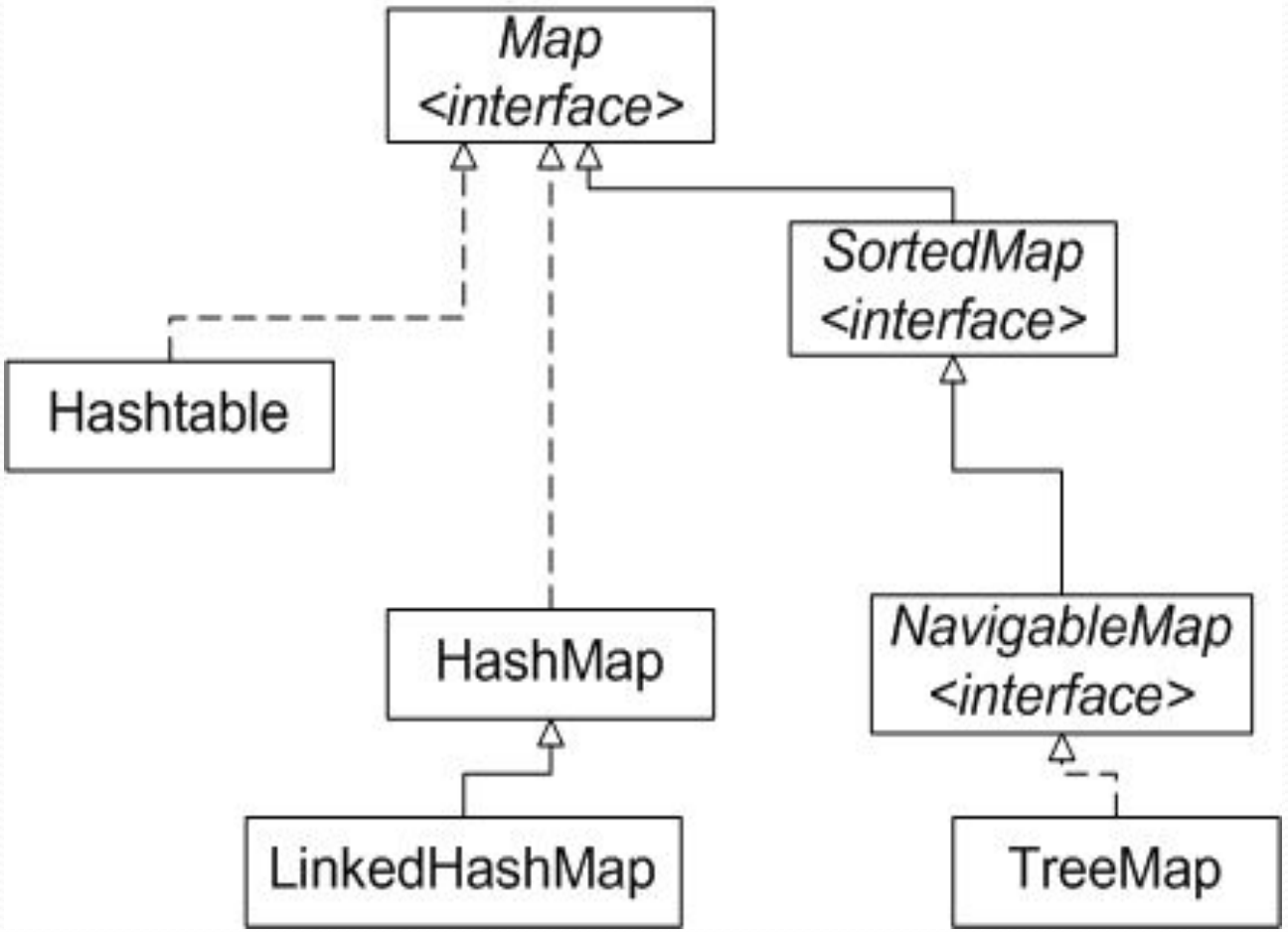
Отображения *Map*

- Отображение представляет собой объект, сохраняющий связи между ключами и значениями в виде пар "***ключ-значение***".
- По заданному ключу можно найти его значение.
- Ключи и значения являются объектами.
- Ключи должны быть уникальными, а значения могут быть дублированными.
- В одних отображениях допускаются ***null*** ключи и ***null*** значения, а в других - они не допускаются.
- Уникальность ключей определяет реализация метода ***equals()***.

Отображения *Map*

- Для корректной работы с картами необходимо переопределить методы ***equals()*** и ***hashCode().***
- Допускается добавление объектов без переопределения этих методов, но найти эти объекты в ***Map*** вы не сможете.

Отображения *Map*



Интерфейсы, поддерживающие отображения

- ***Map*** - отображает уникальные ключи на значения.
- ***Map.Entry*** - описывает элемент карты (пару "ключ значение"). Это вложенный класс ***Map***.
- ***SortedMap*** - расширяет ***Map*** таким образом, что ключи располагаются в порядке по возрастанию.
- ***NavigableMap*** - расширяет ***SortedMap*** для обработки извлечения элементов на основе поиска по ближайшему соответствию.

Интерфейс ***Map***

- Интерфейс ***Map*** отображает уникальные ключи на значения.
- Ключ это объект, который вы используете для последующего извлечения данных.
- Задавая ключ и значение, вы можете помещать значения в объект ***Map***. После того, как это значение сохранено, вы можете получить его по ключу.
- ***Map*** обобщенный интерфейс: ***interface Map<K, V>***
- Здесь ***K*** указывает тип ключей, а ***V*** тип хранимых значений.

Интерфейс ***Map.Entry***

- Интерфейс ***Map.Entry*** описывает элемент карты (пару "ключ значение"). Это вложенный класс ***Map***.

Методы интерфейса ***Map***

- *void clear()*
- Удаляет все пары "ключ-значение" из вызывающей карты.

Методы интерфейса ***Map***

- *boolean containsKey(Object k)*
- Возвращает *true*, если вызывающая карта содержит ключ *k*. В противном случае возвращает *false*.

Методы интерфейса ***Map***

- *boolean containsValue (Object v)*
- Возвращает *true*, если вызывающая карта содержит значение *v*. В противном случае возвращает *false*.

Методы интерфейса ***Map***

- *boolean isEmpty()*
- Возвращает *true*, если вызывающая карта пуста. В противном случае возвращает *false*.

Методы интерфейса *Map*

- *V put(K k, V v)*
- Помещает элемент в вызывающую карту, перезаписывая любое предшествующее значение, ассоциированное с ключом. Ключ и значение это *k* и *v* соответственно. Возвращает *null*, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом.

Методы интерфейса ***Map***

- *void putAll(Map<? extends K, ? extends V> m)*
- Помещает все значения из m в карту.

Методы интерфейса ***Map***

- *V get(Object K)*
- Возвращает значение, ассоциированное с ключом *k*. Возвращает *null*, если ключ не найден.

Методы интерфейса *Map*

- *Set<K> keySet()*
- Возвращает *Set*, который содержит ключи вызывающей карты. Этот метод представляет ключи вызывающей карты в виде набора.

Методы интерфейса *Map*

- *Collection<V> values()*
- Возвращает коллекцию, содержащую значения карты. Этот метод представляет значения, содержащихся в карте, в виде коллекции.

Методы интерфейса *Map*

- *Set<Map. Entry<K, V> entrySet()*
- Возвращает *Set*, содержащий все значения карты. Набор содержит объекты типа *Map.Entry*. То есть этот метод представляет карту в виде набора.

Методы интерфейса ***Map***

- *V remove(Object k)*
- Удаляет элемент, чей ключ равен *k*.

Методы интерфейса ***Map***

- *int size()*
- Возвращает число пар "ключ-значение" в карте.

Методы интерфейса *Map*

Метод	Описание
void clear()	Удаляет все пары "ключ-значение" из вызывающей карты.
boolean containsKey(Object k)	Возвращает ***true***, если вызывающая карта содержит ключ *k*. В противном случае возвращает ***false***.
boolean containsValue (Object v)	Возвращает **true**, если вызывающая карта содержит значение *v*. В противном случае возвращает ***false***.
Set<Map. Entry<K, V> entrySet()	Возвращает ***Set***, содержащий все значения карты. Набор содержит объекты типа ***Map.Entry***. То есть этот метод представляет карту в виде набора.
V get(Object K)	Возвращает значение, ассоциированное с ключом *k*. Возвращает ***null***, если ключ не найден.
boolean isEmpty()	Возвращает ***true***, если вызывающая карта пуста. В противном случае возвращает ***false***.

Методы интерфейса *Map*

Метод	*Описание*
Set<K> keySet()	Возвращает ***Set***, который содержит ключи вызывающей карты. Этот метод представляет ключи вызывающей карты в виде набора.
V put(K k, V v)	Помещает элемент в вызывающую карту, перезаписывая любое предшествующее значение, ассоциированное с ключом. Ключ и значение это *k* и *v* соответственно. Возвращает **null**, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом.
void putAll(Map<? extends K, ? extends V> m)	Помещает все значения из m в карту.

Методы интерфейса *Map*

Метод	*Описание*
V remove(Object k)	Удаляет элемент, чей ключ равен *k*.
int size()	Возвращает число пар "ключ-значение" в карте.
Collection<V> values()	Возвращает коллекцию, содержащую значения карты. Этот метод представляет значения, содержащихся в карте, в виде коллекции.

Класс *HashMap*

- Класс ***HashMap*** реализует интерфейс ***Map***. Он использует хеш-таблицу для хранения карты.
- Это позволяет обеспечить константное время выполнения методов ***get()*** и ***put()*** даже при больших наборах.
- Ключи и значения могут быть любых типов, в том числе и **null**.
- ***HashMap*** обобщенный класс со следующим объявлением:

class HashMap<K, V>

Пример использования класса *HashMap*

```
public class HashMapDemo {
    public static void main(String[] args) {
        Map<String, Double> hashMap = new HashMap<>();

        hashMap.put("Иванов", 3434.34);
        hashMap.put("Петров", 123.22);
        hashMap.put("Сидоров", 1378.00);

        Set<String> keys = hashMap.keySet();

        for (String key : keys) {
            System.out.print(key + ": ");
            System.out.println(hashMap.get(key));
        }
    }
}
```

Интерфейс ***SortedMap***

- Интерфейс ***SortedMap*** расширяет ***Map***. Он гарантирует, что элементы размещаются в возрастающем порядке значений ключей.

Методы ***SortedMap***

- *Comparator<? super K> comparator()*
- Возвращает компаратор вызывающей сортированной карты. Если картой используется естественный порядок, возвращается *null*.

Методы ***SortedMap***

- *K firstKey()*

Возвращает первый ключ вызывающей карты.

- *K lastKey()*

Возвращает последний ключ вызывающей карты.

Методы ***SortedMap***

- *SortedMap<K, V> headMap(K end)*

Возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых меньше *end*.

- *SortedMap<K, V> subMap(K start, K end)*

Возвращает карту, содержащую элементы вызывающей карты, чей ключ больше или равен *start* и меньше *end*.

- *SortedMap<K, V> tailMap(K start)*

Возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых больше *start*.

Методы *SortedMap*

Метод	*Описание*
Comparator<? super K> comparator()	Возвращает компаратор вызывающей сортированной карты. Если картой используется естественный порядок, возвращается ***null***.
K firstKey()	Возвращает первый ключ вызывающей карты.
K lastKey()	Возвращает последний ключ вызывающей карты.
SortedMap<K, V> headMap(K end)	Возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых меньше *end*.
SortedMap<K, V> subMap(K start, K end)	Возвращает карту, содержащую элементы вызывающей карты, чей ключ больше или равен *start* и меньше *end*.
SortedMap<K, V> tailMap(K start)	Возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых больше *start*.

Интерфейс *NavigableMap*

- Интерфейс ***NavigableMap*** был добавлен в Java 6. Она расширяет ***SortedMap*** и определяет поведение карты, поддерживающей извлечение элементов на основе ближайшего соответствия заданному ключу или ключам.

Методы *NavigableMap*

- *Map.Entry<K,V>* ***lowerEntry****(K key)*
- *Map.Entry<K,V>* ***floorEntry****(K key)*
- *Map.Entry<K,V>* ***higherEntry****(K key)*
- *Map.Entry<K,V>* ***ceilingEntry****(K key)*
- Методы позволяют получить соответственно меньший, меньше или равный, больший, больше или равную пару “ключ-значение” по отношению к заданному.

Методы *NavigableMap*

- *K **lowerKey**(K key)*
- *K **floorKey**(K key)*
- *K **higherKey**(K key)*
- *K **ceilingKey**(K key)*
- Методы позволяют получить соответственно меньший, меньше или равный, больший, больше или равный ключ по отношению к заданному.

Методы ***NavigableMap***

- *Map.Entry<K,V>* ***pollFirstEntry()***
- *Map.Entry<K,V>* ***pollLastEntry()***
- *Map.Entry<K,V>* ***firstEntry()***
- *Map.Entry<K,V>* ***lastEntry()***
- Методы ***pollFirstEntry*** и ***pollLastEntry*** возвращают соответственно первый и последний элементы карты, удаляя их из коллекции. Методы ***firstEntry*** и ***lastEntry*** также возвращают соответствующие элементы, но без удаления.

Методы ***NavigableMap***

- *NavigableMap<K,V>* ***descendingMap()***
- Возвращает карту, отсортированную в обратном порядке.

Методы ***NavigableMap***

- *NavigableSet<**K**>* ***navigableKeySet****()*
- *NavigableSet<K>* ***descendingKeySet****()*
- Методы, позволяющие получить набор ключей, отсортированных в прямом и обратном порядке соответственно.

Методы ***NavigableMap***

- *NavigableMap<K,V>* ***subMap****(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)*
- *NavigableMap<K,V>* ***headMap****(K toKey, boolean inclusive)*
- *NavigableMap<K,V>* ***tailMap****(K fromKey, boolean inclusive)*
- Методы, позволяющие извлечь из карты подмножество. Как и в случае ***NavigableSet***, указываем в параметрах начальный и конечный элементы массива ключей, а также необходимость включения в выходной набор граничных элементов

Методы *NavigableMap*

Метод	Описание
Map.Entry<K,V> ***lowerEntry****(K key)*	Методы позволяют получить соответственно меньший, меньше или равный, больший, больше или равную пару “ключ-значение” по отношению к заданному.
Map.Entry<K,V> ***floorEntry****(K key)*	
Map.Entry<K,V> ***higherEntry****(K key)*	
Map.Entry<K,V> ***ceilingEntry****(K key)*	
K ***lowerKey****(K key)*	Методы позволяют получить соответственно меньший, меньше или равный, больший, больше или равный ключ по отношению к заданному.
K ***floorKey****(K key)*	
K ***higherKey****(K key)*	
K ***ceilingKey****(K key)*	

Методы *NavigableMap*

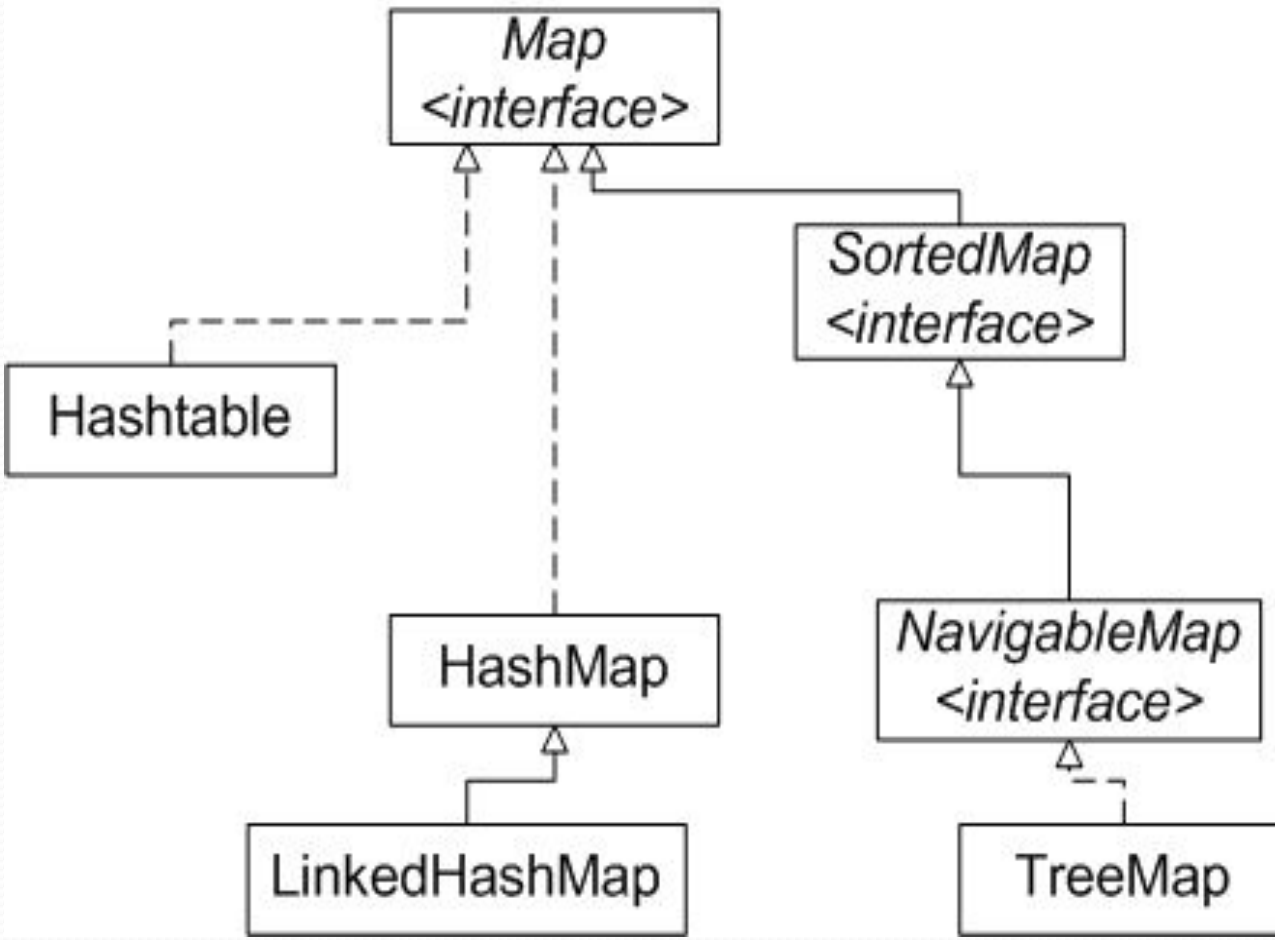
Метод	Описание
Map.Entry<K,V> ***pollFirstEntry****()*	Методы ***pollFirstEntry*** и ***pollLastEntry*** возвращают соответственно первый и последний элементы карты, удаляя их из коллекции. Методы ***firstEntry*** и ***lastEntry*** также возвращают соответствующие элементы, но без удаления.
Map.Entry<K,V> ***pollLastEntry****()*	
Map.Entry<K,V> ***firstEntry****()*	
Map.Entry<K,V> ***lastEntry****()*	
NavigableMap<K,V> ***descendingMap****()*	Возвращает карту, отсортированную в обратном порядке.
NavigableSet<K> ***navigableKeySet****()*	Методы, позволяющие получить набор ключей, отсортированных в прямом и обратном порядке соответственно.
NavigableSet<K> ***descendingKeySet****()*	

Методы *NavigableMap*

Методы	Описание
NavigableMap<K,V> ***subMap****(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)*	Методы, позволяющие извлечь из карты подмножество. Как и в случае ***NavigableSet***, указываем в параметрах начальный и конечный элементы массива ключей, а также необходимость включения в выходной набор граничных элементов.
NavigableMap<K,V> ***headMap****(K toKey, boolean inclusive)*	
NavigableMap<K,V> ***tailMap****(K fromKey, boolean inclusive)*	

Классы отображений

- ***HashMap*** - для использования хеш-таблицы.
- ***TreeMap*** - для использования дерева.
- ***LinkedHashMap*** - расширяет ***HashMap***, разрешая итераторы в порядке вставки.



Класс *TreeMap*

- ***TreeMap*** – хранит элементы в порядке сортировки.
- ***TreeMap*** сортирует элементы по возрастанию от первого к последнему.
- Порядок сортировки может задаваться реализацией интерфейсов ***Comparator*** и ***Comparable***.
- Реализация ***Comparator*** передается в конструктор ***TreeMap***, ***Comparable*** используется при добавлении элемента в карту.

Конструкторы класса *TreeMap*

- ***TreeMap()***
- ***TreeMap(Comparator<? super K> comp)***
- ***TreeMap(Map<? extends K, ? extends V> m)***
- ***TreeMap(SortedMap<K, ? extends V> sm)***

Пример класса *TreeMap*

```
public class TreeMapDemo {
    public static void main(String[] args) {
        SortedMap<String, Double> treeMap =new TreeMap<>();

        treeMap.put("Иванов", 3434.34);
        treeMap.put("Петров", 123.22);
        treeMap.put("Сидоров", 1378.00);

        treeMap.forEach((k, v) -> System.out.println(k + ": " + v));
    }
}
```

Класс *LinkedHashMap*

- Класс ***LinkedHashMap*** расширяет ***HashMap***. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать итерацию по карте в порядке вставки.

Изменяемые объекты в качестве ключа

```
public class ProductKeyDemo {
    public static void main(String[] args) {
        Map<Product, String> map = new HashMap<>();
        Product doll = new Product("Кукла", 534, "Украина");
        Product box = new Product("Кубик", 34, "Украина");
        Product car = new Product("Машинка", 200, "Украина");

        map.put(doll, "Антошка");
        map.put(box, "Антошка");
        map.put(car, "Детский мир");

        System.out.println(map.get(doll));
        doll.setCost(434);

        System.out.println(map.get(doll));
    }
}
```

Пример “backed Collections”

```
public class BackedCollections {
    public static void main(String[] args) {
        SortedMap<String, String> map =new TreeMap<>();
        map.put("а", "арбуз");
        map.put("в", "вишня");
        map.put("д", "дыня");

        SortedMap<String, String> subMap = map.subMap("б", "ж");
        System.out.println(map + " " + subMap);

        map.put("б", "брусника");
        subMap.put("г", "груша");
        map.put("я", "яблоко");
        //   subMap.put("с", "слива");
        System.out.println(map + " " + subMap);
    }
}
```

Array-backed списки

```
public class BackedArrayCollection {
    public static void main(String[] args) {
        String[] array = {"арбуз", "вишня", "дыня"};
        List<String> list = Arrays.asList(array);

        System.out.println("Список:" + list);
        System.out.println("Массив:" + Arrays.toString(array));

        list.set(0, "яблоко");
        array[1] = "брусника";

        System.out.println("Список:" + list);
        System.out.println("Массив:" + Arrays.toString(array));
    }
}
```

Алгоритмы коллекций

- Каркас коллекций определяет несколько алгоритмов, которые могут быть применимы к коллекциям и картам. Эти алгоритмы определены как статические методы в классе ***Collections***.

Пример метода ***sort(List)***

```
public class SortCollections {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("красный", "синий", "зеленый");
        System.out.println("Перед сортировкой: " + list);
        Collections.sort(list);
        System.out.println("После сортировки: " + list);
        Collections.sort(list, Collections.reverseOrder());
        System.out.println("После обратной сортировки: " + list);
    }
}
```

Пример метода *binarySearch(List)*

```
public class BinarySearchDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("красный", "синий", "зеленый");
        Collections.sort(list);

        System.out.println(list);
        System.out.println(Collections.binarySearch(list, "красный"));
        System.out.println(Collections.binarySearch(list, "черный"));
    }
}
```

Пример методов *reverse(List), shuffle(List)*

```
public class CollectionsExample1 {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("красный", "синий", "зеленый",
        System.out.println("Перед reversing: " + list);
        Collections.reverse(list);
        System.out.println("После reversing: " + list);
        Collections.shuffle(list);
        System.out.println("После shuffling: " + list);
    }
}
```

Пример метода *fill(List, Object)*

```
public class CollectionsFillDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("красный", "синий", "зеленый");
        Collections.fill(list, "черный");
        System.out.println(list);
    }
}
```

Пример методов *max(List), min(List)*

```
public class CollectionsMinMaxDemo {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(2, 2, 5, 8, 9);
        System.out.println(Collections.max(list));
        System.out.println(Collections.min(list));
    }
}
```

Пример метода *copy(List, List)*

```
public class CollectionsCopyDemo {
    public static void main(String[] args) {
        List<Integer> src = Arrays.asList(1, 2, 3);
        List<Integer> dest = Arrays.asList(4, 5, 6, 7);
        Collections.copy(dest, src);
        System.out.println(dest);
    }
}
```

Пример метода *rotate(List, int)*

```
public class CollectionsRotateDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("a", "b", "c", "d", "e");
        System.out.println(list);

        Collections.rotate(list, 2);
        System.out.println(list);

        Collections.rotate(list, -1);
        System.out.println(list);
    }
}
```

Пример метода *checkedCollection()*

```
public class MyCheckedCollection {
    public static void main(String[] a) {
        List myList = new ArrayList();
        myList.add("one");
        myList.add("two");
        myList.add("three");
        myList.add("four");
        Collection checkList =
                Collections.checkedCollection(myList, String.class);
        System.out.println("Checked list content: " + checkList);
        myList.add(10);
        checkList.add(10); //throws ClassCastException
    }
}
```

Пример метода *frequency()*

```
public class CollectionsFrequencyDemo {
    public static void main(String[] args) {
        Collection<String> collection = Arrays.asList("red", "cyan", "red"
        System.out.println(Collections.frequency(collection, "red"));
    }
}
```

Методы *Collections*

Методы	*Назначение*
sort(List)	Сортировать список.
binarySearch(List, Object)	Бинарный поиск элементов в списке.
reverse(List)	Изменить порядок элементов в списке на противоположный.
shuffle(List)	Случайно перемешать элементы.
fill(List, Object)	Заменить каждый элемент заданным.
copy(List dest, List src)	Скопировать список src в dst.
min(Collection)	Вернуть минимальный элемент коллекции.
max(Collection)	Вернуть максимальный элемент коллекции.
rotate(List list, int distance)	Циклически повернуть список на указанное число элементов.

Методы *Collections*

Методы	*Назначение*
replaceAll*(List list, Object oldVal, Object newVal)*	Заменить все объекты на указанные.
indexOfSubList*(List source, List target)*	Вернуть индекс первого подсписка source, который эквивалентен target.
lastIndexOfSubList*(List source, List target)*	Вернуть индекс последнего подсписка source, который эквивалентен target.
swap*(List, int, int)*	Заменить элементы в указанных позициях списка.
unmodifiableCollection *(Collection)*	Создает неизменяемую копию коллекции. Существуют отдельные методы для ***Set, List, Map***, и т.д.
synchronizedCollection *(Collection)*	Создает потокобезопасную копию коллекции. Существуют отдельные методы для ***Set, List, Map***, и т.д.

Методы ***Collections***

Методы	***Назначение***
checkedCollection*(Collection<E> c, Class<E> type)*	Создает типо-безопасную копию коллекции, предотвращая появление неразрешенных типов в коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
<T> Set<T> ***singleton****(T o);*	Создает неизменяемый ***Set***, содержащую только заданный объект. Существуют методы для ***List*** и ***Map***.
<T> List<T> ***nCopies****(int n, T o)*	Создает неизменяемый ***List***, содержащий n копий заданного объекта.
frequency*(Collection , Object)*	Подсчитать количество элементов в коллекции.
reverseOrder()	Вернуть ***Comparator***, которые предполагает обратный порядок сортировки элементов.

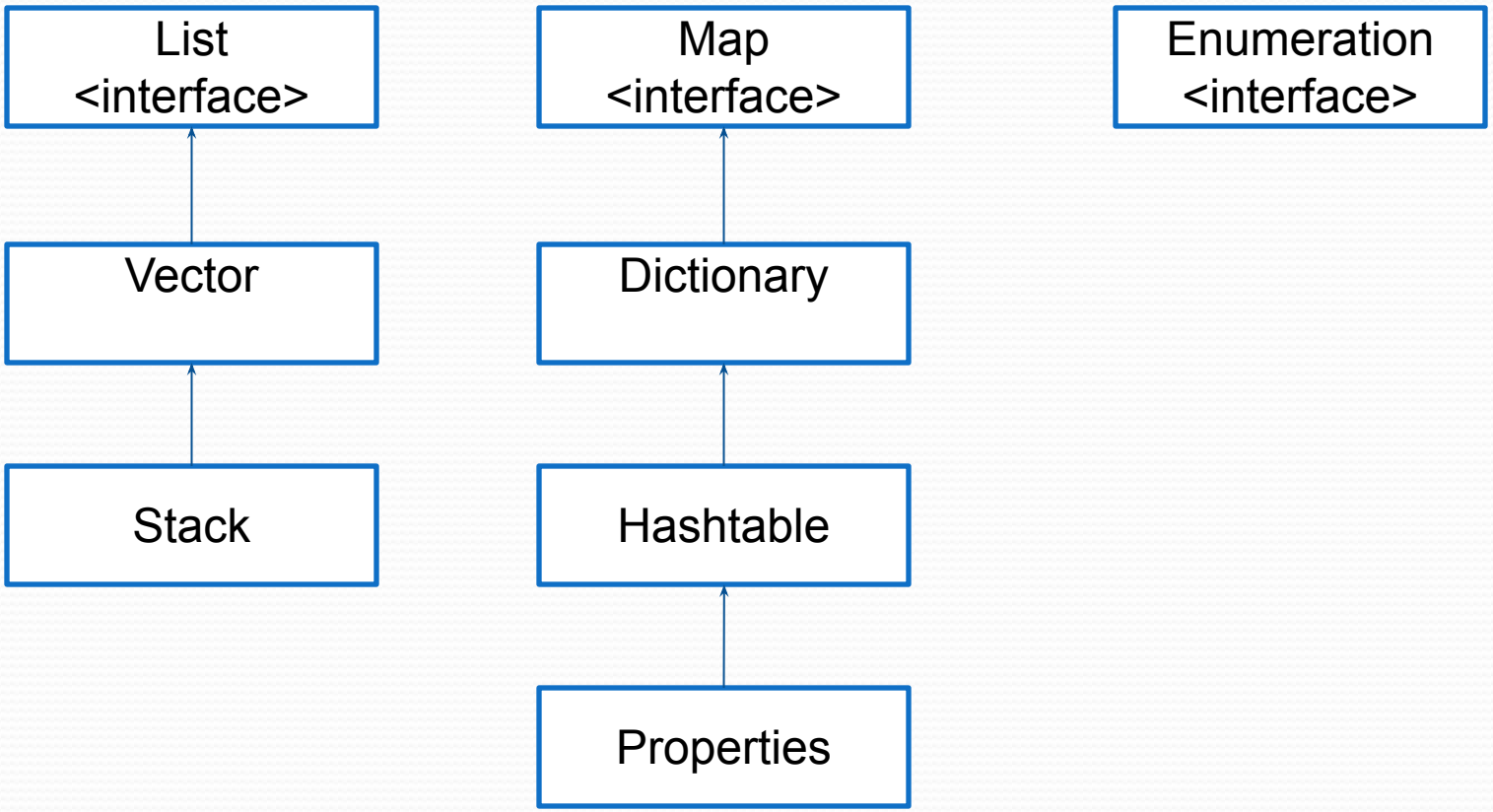
Методы *Collections*

Методы	*Назначение*
list*(Enumeration<T> e)*	Вернуть ***Enumeration*** в виде ***ArrayList***.
disjoint*(Collection, Collection)*	Определить, что коллекции не содержат общих элементов.
addAll*(Collection<? super T>, T[])*	Добавить все элементы из массива в коллекцию
newSetFromMap*(Map)*	Создать ***Set*** из ***Map***.
asLifoQueue*(Deque)*	Создать LIFO ***Queue*** представление из ***Deque***.

Унаследованные коллекции

- Унаследованные коллекции (***Legacy Collections***) – это коллекции языка Java 1.0/1.1.
- И хотя эти классы были достаточно удобны, им недоставало общей, объединяющей основы.
- В ряде распределенных приложений до сих пор применяются унаследованные коллекции, более медленные в обработке, но при этом потокобезопасные, существовавшие в языке Java с момента его создания.

Унаследованные коллекции



Класс *Vector*

- Класс ***Vector*** реализует динамический массив.
- Он подобен ***ArrayList***, но с двумя отличиями: ***Vector*** синхронизирован и включает много унаследованных методов, не являющихся частью каркаса коллекций.
- С появлением коллекций ***Vector*** был перепроектирован как расширение ***AbstractList***, и в него была добавлена реализация интерфейса ***List***.
- В версии jDK5 он был перепроектирован под применение обобщённого синтаксиса, и в нем появилась реализация интерфейса ***Iterable***.

Класс ***Stack***

- ***Stack*** это подкласс ***Vector***, который реализует стандартный стек LIFO.
- ***Stack*** определяет только конструктор по умолчанию, создающий пустой стек. С появлением версии Java 5 подкласс ***Stack*** был перепроектирован под обобщенный синтаксис, и теперь он объявлен следующим образом: ***class Stack<E>***

Класс *Dictionary*

- ***Dictionary*** это абстрактный класс, представляющий репозиторий для хранения пар "ключ-значение" и работающий в основном подобно ***Map***. Передав ключ и значение, вы можете сохранить значение в объекте ***Dictionary***.
- Однажды сохраненное значение можно извлечь по его ключу. То есть, подобно карте, ***Dictionary*** (словарь) можно считать списком пар "ключ-значение".
- Хотя пока ***Dictionary*** не объявлен нежелательным, его можно рассматривать как устаревший, поскольку его полностью заменяет ***Map***.
- С появлением Java 5 класс ***Dictionary*** был также сделан обобщенным: ***class Dictionary<K, V>***

Класс ***Hashtable***

- ***Hashtable*** это часть исходного пакета *java.util* и конкретная реализация ***Dictionary***.
- Однако с появлением коллекций класс ***Hashtable*** был перепроектирован с тем, чтобы также реализовывать интерфейс ***Map***. То есть ***Hashtable*** теперь интегрирован в каркас коллекций. Он подобен ***HashMap***, но синхронизирован.
- Подобно ***HashMap***, ***Hashtable*** сохраняет пары "ключ-значение" в хеш-таблице. Однако ни ключи, ни значения не могут быть равны ***null***.
- ***Hashtable*** был сделан обобщенным в Java 5:
- ***class Hashtable<K, V>***

Класс ***Properties***

- ***Properties*** (свойства) подкласс ***Hashtable***. Он служит для поддержки списков значений, в которых ключами являются объекты ***String*** и значения также объекты ***String***.
- Класс ***Properties*** используется многими другими классами Java.
- Одно удобное свойство класса ***Properties*** это то, что вы можете указать значения по умолчанию, которые будут возвращены, если никакое значение не ассоциировано с определенным ключом.
- Например, значение по умолчанию может быть указано вместе с ключом в методе ***getProperty()*** как, например, в ***getProperty***("имя", "значение по умолчанию"). Если значение "имя" не найдено, возвращается "значение по умолчанию".

Пример использования класса *Properties*

```
public class PropertyDemo {
    public static void main(String[] args) {
        Properties capitals = new Properties();

        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");

        // Get a set-view of the keys.
        Set states = capitals.keySet();

        // Show all of the states and capitals.
        for (Object name : states) {
            System.out.println("The capital of " + name + " is "
                    + capitals.getProperty((String) name) + ".");
        }

        System.out.println();

        // Look for state not in list -- specify default.
        String str = capitals.getProperty("Florida", "Not Found");
```

Интерфейс ***Enumeration***

- Интерфейс ***Enumeration*** определяет методы, которыми вы можете перечислить (получая по одному за раз) элементы в коллекции объектов.
- Этот унаследованный интерфейс был замещен ***Iterator***.
- Хотя ***Enumeration*** и не является не рекомендованным, но считается устаревшим для нового кода.
- Однако он используется несколькими методами унаследованных классов (таких как ***Vector*** или ***Properties***), также некоторыми другими классами API.
- Поскольку он все еще задействован, он был перепроектирован в обобщенном виде jDK 5. Он имеет следующее объявление:

interface Enumeration<E>

Интерфейс ***Enumeration***

- В ***Enumeration*** определены следующие два метода:

boolean ***hasMoreElements****()*
E ***nextElement****()*