|  |  |  |
| --- | --- | --- |
|  | Основная функциональность | Примеры типичного использования |
| **Set** | Множество неповторяющихся объектов. Разрешено наличие только одной ссылки типа null. Поддерживает добавление, удаление, проверку наличия элемента, выгрузку в массив.  Интерфейс Set унаследован от интерфейса Collection, но никаких новых методов не добавляет. Изменяется только смысл метода add(E item) — он не станет добавлять объект item, если он уже присутствует во множестве. Некоторые основные методы интерфейса Set   * **boolean add (E item)**: добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном - false * **void clear ()**: удаляет все элементы из коллекции * **boolean contains (Object item)**: возвращает true, если объект item содержится в коллекции, иначе возвращает false * **boolean isEmpty ()**: возвращает true, если коллекция пуста, иначе возвращает false * **boolean remove (Object item)**: возвращает true, если объект item удачно удален из коллекции, иначе возвращается false * **int size ()**: возвращает число элементов в коллекции * **Object[] toArray ()**: возвращает массив, содержащий все элементы коллекции * **Iterator<E> iterator()**: возвращает итератор коллекции | **HashSet** - коллекция, не позволяющая хранить одинаковые объекты(как и любой Set**).  HashSet** инкапсулирует в себе объект **HashMap** (то-есть использует для хранения хэш-таблицу). Хеш-таблица хранит информацию, используя так называемый механизм хеширования, в котором содержимое ключа используется для определения уникального значения, называемого хеш-кодом. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов add(), contains(), **remove()** и **size()** , даже для больших наборов.  При использовании ***HashSet*** для хранения объектов собственных классов, необходимо переопределить методы hashCode() и equals()*,* иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод hashCode() класса Object (который скорее всего вернет разный хэш-код для этих объектов). Класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов.  Пример работы с HashSet:  package collectionapp;    import java.util.HashSet;    public class CollectionApp {        public static void main(String[] args) {            HashSet<String> states = new HashSet<String>();            // добавим в список ряд элементов          states.add("Германия");          states.add("Франция");          states.add("Италия");            System.out.printf("В списке %d элементов \n", states.size());            for(String state : states){                System.out.println(state);          }            states.remove("Германия");            HashSet<Person> people = new HashSet<Person>();          people.add(new Person("Mike"));          people.add(new Person("Tom"));          people.add(new Person("Nick"));          for(Person p : people){                System.out.println(p.getName());          }      }  }  class Person{        private String name;      public Person(String value){            name=value;      }      String getName(){return name;}  }  **LinkedHashSet** -  поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. То есть, когда идет перебор объекта класса LinkedHashSet с применением итератора, элементы извлекаются в том порядке, в каком они были добавлены.  **TreeSet** - коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet хорош тем, что для операций add, remove и contains потребуется гарантированное время log(n). |
| **List** | Упорядоченный список. Объекты хранятся в порядке их добавления в список. Доступ к элементам списка осуществляется по индексу.  Поддерживает добавление элементов в любое место списка, удаление, изменение, поиск элементов, выгрузку в массив. Некоторые основные методы интерфейса List, которые часто используются в ArrayList:   * **void add(int index, E obj)**: добавляет в список по индексу index объект obj * **boolean addAll(int index, Collection<? extends E> col)**: добавляет в список по индексу index все элементы коллекции col. Если в результате добавления список был изменен, то возвращается true, иначе возвращается false * **E get(int index)**: возвращает объект из списка по индексу index * **int indexOf(Object obj)**: возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1 * **int lastIndexOf(Object obj)**: возвращает индекс последнего вхождения объекта obj в список. Если объект не найден, то возвращается -1 * **E remove(int index)**: удаляет объект из списка по индексу index, возвращая при этом удаленный объект * **E set(int index, E obj)**: присваивает значение объекта obj элементу, который находится по индексу index * **void sort(Comparator<? super E> comp)**: сортирует список с помощью компаратора comp * **List<E> subList(int start, int end)**: получает набор элементов, которые находятся в списке между индексами start и end | **ArrayList** - пожалуй самая часто используемая коллекция. ArrayList инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов. Так как ArrayList использует массив, то  время доступа к элементу по индексу минимально (В отличии от LinkedList). При удалении произвольного элемента из списка, все элементы находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, capacity) не изменяется. Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером (n \* 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент.  Пример работы с ArrayList:  package com.friendlyfunction.collections;  import java.util.ArrayList;  import java.util.List;  public class ArrayListExample {  public static void main(String[] args){  List<String> list = new ArrayList<String>();  list.add("Молоко");  list.add("Кока-Кола");  list.add("Хлеб");  list.add("Репа");  list.add("Хамон");  System.out.println("Список: " + list);  list.remove("Хамон");  System.out.println("Список: " + list);  list.set(1, "Вятский квас");  System.out.println("Список: " + list);    }  }  **LinkedList** - Двусвязный список. Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и  две ссылки («связки») на следующий и предыдущий узел списка. Доступ к произвольному элементу осуществляется за линейное время (но доступ к первому и последнему элементу списка всегда осуществляется за константное время — ссылки постоянно хранятся на первый и последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента). В целом же, LinkedList в абсолютных величинах проигрывает ArrayList и по потребляемой памяти и по скорости выполнения операций. |
| **Queue** | Коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. Предоставляет дополнительные операции вставки, получения и контроля.  Поддерживает добавление, чтение и извлечение (с удалением) элементов с разных концов очереди, а также поиск элементов. Свою функциональность он раскрывает через следующие методы:   * **E element()**: возвращает, но не удаляет, элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException * **boolean offer(E obj)**: добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false * **E peek()**: возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null * **E poll()**: возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null * **E remove()**: возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException   Интерфейс Deque расширяет вышеописанный интерфейс Queue и определяет поведение двунаправленной очереди, которая работает как обычная однонаправленная очередь, либо как стек, действующий по принципу LIFO (последний вошел - первый вышел).  Интерфейс Deque определяет следующие методы:   * **void addFirst(E obj)**: добавляет элемент в начало очереди * **void addLast(E obj)**: добавляет элемент obj в конец очереди * **E getFirst()**: возвращает без удаления элемент из головы очереди. Если очередь пуста, генерирует исключение NoSuchElementException * **E getLast()**: возвращает без удаления последний элемент очереди. Если очередь пуста, генерирует исключение NoSuchElementException * **boolean offerFirst(E obj)**: добавляет элемент obj в самое начало очереди. Если элемент удачно добавлен, возвращает true, иначе - false * **boolean offerLast(E obj)**: добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false * **E peekFirst()**: возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null * **E peekLast()**: возвращает без удаления последний элемент очереди. Если очередь пуста, возвращает значение null * **E pollFirst()**: возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null * **E pollLast()**: возвращает с удалением последний элемент очереди. Если очередь пуста, возвращает значение null * **E pop()**: возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException * **void push(E element)**: добавляет элемент в самое начало очереди * **E removeFirst()**: возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException | **PriorityQueue** - единственная прямая реализация интерфейса **Queue** (не считая **LinkedList**, который больше является списком, чем очередью). Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс **Comparable**), либо с помощью интерфейса **Comparator**, полученному в конструкторе.  **ArrayDeque** — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Эта коллекция представляет собой реализацию с использованием массивов, подобно ArrayList, но не позволяет обращаться к элементам по индексу и хранение null. Как заявлено в документации, коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO.  Пример работы с ArrayDeque:  import java.util.ArrayDeque;    public class CollectionApp {        public static void main(String[] args) {            ArrayDeque<String> states = new ArrayDeque<String>();          // стандартное добавление элементов          states.add("Германия");          states.add("Франция");          // добавляем элемент в самое начало          states.push("Великобритания");            // получаем первый элемент без удаления          String sFirst = states.getFirst();          String sLast = states.getLast();            while(states.peek()!=null){              // извлечение c начала              System.out.println(states.pop());          }          System.out.printf("Размер очереди: %d \n", states.size());              ArrayDeque<Person> people = new ArrayDeque<Person>();          people.addFirst(new Person("Tom"));          people.addLast(new Person("Nick"));          for(Person p : people){                System.out.println(p.getName());          }      }  }  class Person{        private String name;      public Person(String value){            name=value;      }      String getName(){return name;}  } |
| **Map** | Ассоциативный массив вида ключ=>значение, где ключ и значение могут быть объектами любого класса. Ключи в списке должны быть уникальными.  Поддерживает добавление и удаление пар ключ=>значение, поиск ключей и значений, перезапись значений ключей, выгрузку в коллекцию. Все классы отображений реализуют обобщенный интерфейс Map<K, V>, который определяет основную функциональность через следующие методы:   * **void clear()**: очищает коллекцию * **boolean containsKey(Object k)**: возвращает true, если коллекция содержит ключ k * **boolean containsValue(Object v)**: возвращает true, если коллекция содержит значение v * **Set<Map.Entry<K, V>> entrySet()**: возвращает набор элементов коллекции. Все элементы представляют объект Map.Entry * **boolean equals(Object obj)**: возвращает true, если коллекция идентична коллекции, передаваемой через параметр obj * **boolean isEmpty()**: возвращает true, если коллекция пуста * **V get(Object k)**: возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение null * **V put(K k, V v)**: помещает в коллекцию новый объект с ключом k и значением v. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение null * **Set<K> keySet()**: возвращает набор всех ключей отображения * **Collection<V> values()**: возвращает набор всех значений отображения * **void putAll(Map<? extends K, ? extends V> map)**: добавляет в коллекцию все объекты из отображения map * **V remove(Object k)**: удаляет объект с ключом k * **int size()**: возвращает количество элементов коллекции | **HashMap** — основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени.  Пример работы с HashMap:  package com.friendlyfunction.collections;  import java.util.HashMap;  import java.util.Map;  import java.util.Map.Entry;  public class HashMapTutorial {  public static void main(String[] args) {  Map<String, Integer> map = new HashMap<String, Integer>();  map.put("яблоко", 4);  map.put("апельсин", 2);  map.put("тыква", 1);  System.out.println("апельсин: " + map.get("апельсин"));  System.out.println(map);  map.put("помидор", 3);  for(Entry<String, Integer> e : map.entrySet()){  System.out.println(e.getValue() + " " + e.getKey());  }  map.remove("апельсин");  System.out.println(map);  }  }  **LinkedHashMap** -  расширяет класс **HashMap**. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать перебор карты в порядке вставки. То есть, когда происходит итерация по коллекционному представлению объекта класса LinkedHashMap, элементы будут возвращаться в том порядке, в котором они вставлялись. Вы также можете создать объект класса LinkedHashMap, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ  **TreeMap** - расширяет класс **AbstractMap** и реализует интерфейс **NavigatebleMap.** Он создает коллекцию, которая для хранения элементов применяет дерево*.* Объекты сохраняются в отсортированном порядке по возрастанию*.* Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена. |