БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра веб-технологий и компьютерного моделирования

А. С. Кравчук, А. И. Кравчук, Е. В. Кремень

ЯЗЫК JAVA.
КОЛЛЕКЦИИ, РЕАЛИЗУЮЩИЕ ИНТЕРФЕЙСЫ QUEUE, DEQUE, MAP, SET

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

МИНСК
2023

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1
К78

Утверждено на заседании кафедры
веб-технологий и компьютерного моделирования
8 ноября 2022 г., протокол № 3

Р е ц е н з е н т
кандидат физико-математических наук,
доцент *Г. А. Расолько*

Кравчук, А. С.

К78 Язык Java. Коллекции, реализующие интерфейсы Queue, Deque, Map, Set : учеб. материалы / А. С. Кравчук, А. И. Кравчук, Е. В. Кремень. – Минск : БГУ, 2023. – 78 с.

Рассматриваются коллекции PriorityQueue, ArrayDeque и LinkedList, реализующие интерфейсы Queue; коллекции Hashtable, HashMap, LinkedHashMap, TreeMap, предоставляемые интерфейсом Map; и коллекции HashSet, LinkedHashSet TreeSet как наиболее часто используемые реализации интерфейса Set. Для каждой коллекции даются пояснения, как пользоваться классами и методами. Приводится необходимый теоретический материал и код программ, что существенно ускоряет усваивание материала, а также способствует более квалифицированному подходу к программированию. Материал по теме тщательно подобран, хорошо структурирован и компактно изложен.

Издание ориентировано как на тех, кто не имеет опыта практического программирования на языке Java, так и на тех, кто хотел бы систематизировать и улучшить свои знания.

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1

© Кравчук А. С., Кравчук А. И.,
Кремень Е. В., 2023
© БГУ, 2023

Оглавление

Интерфейс Queue 5

Методы Queue, дополняющие методы Collection 5

Коллекция PriorityQueue 6

Конструктор без параметров и с одним целочисленным параметром 6

Конструктор, принимающий в качестве значения предварительно инициализированную коллекцию 8

Конструктор, принимающий в качестве значения компаратор 9

Использование синтаксиса анонимного класса для создания Compparator-а 11

Применение нескольких компараторов при сортировке 12

Использование компараторов без создания, имплементирующих интерфейс Comparator классов 14

Использование методов offer(), peek(), poll() интерфейса Queue 15

К вопросу о создании приоритетной очереди из различных типов данных 17

Применение приоритетной очереди для сортировки элементов 20

Перечислим собственно методы коллекции PriorityQueue 21

Интерфейс Deque 22

Методы интерфейса 22

Класс ArrayDeque 24

Использования методов класса ArrayDeque 25

Примеры статических и экземплярных generic-методов для работы с объектом коллекции ArrayDeque 28

Коллекция LinkedList 30

Коллекции, реализующие Deque vs коллекции, реализующие только List 32

Дополнительные примеры методов для работы с коллекциями. Специфика создания generic-методов для обработки коллекций 33
Общие сведения о хэш-таблицах 39
Интерфейс Map 40
Коллекция Hashtable 41
Дополнительные методы Hashtable 44
Выводы по применению методов коллекции Hashtable 46
Коллекция HashMap 47
Коллекция LinkedHashMap 50
Интерфейс SortedMap 57
Интерфейс NavigableMap 57
Коллекция TreeMap 58
Пример использования TreeMap с конструктором без параметров 59
Конструктор с компаратором в качестве параметра 61
Использование конструктора, принимающего в качестве значения объект коллекции, реализующей интерфейс Map 63
Применение конструктора, инициализируемого объектом коллекции TreeMap .. 64
Синхронизация коллекции 66
Интерфейс Set 66
Коллекция HashSet 66
Коллекция LinkedHashSet 69
Интерфейс SortedSet 70
Интерфейс NavigableSet 71
Коллекция TreeSet 71
Пример использования TreeSet с конструктором без параметров 73
Конструктор с компаратором в качестве параметра 74
Литература 77

Интерфейс Queue

Этот интерфейс описывает коллекции с предопределенным способом вставки и извлечения элементов, а именно — очереди (структура данных FIFO).

<u>Замечание.</u>

Очередь – структура данных, в которой добавление элемента возможно лишь в конец очереди, а выборка — только из начала очереди, при этом выбранный элемент из очереди удаляется. Таким образом в очереди реализуется принцип «первым вошел — первым вышел» (англ. first-in, first-out — FIFO).

Помимо методов, определенных в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь. Большинство реализаций данного интерфейса находится в пакете java.util.concurrent.

Методы Queue, дополняющие методы Collection

Список методов интерфейса:

- boolean **add**(E obj) - вставляет элемент в очередь, если очередь заполнена, то генерирует исключение IllegalStateException.
- 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.

Коллекция `PriorityQueue`

`PriorityQueue` — является апосредованной реализацией интерфейса `Queue`, через расширение класса `AbstractCollection`. Особенностью данной очереди является возможность управления порядком элементов (приоритетом).

«По умолчанию», элементы сортируются с использованием «`natural ordering`», но это поведение может быть переопределено при помощи объекта `Comparator`, который задается при создании очереди. Данная коллекция ***не*** поддерживает `null` в качестве элементов.

Коллекция поддерживает пять конструкторов.

Конструктор без параметров и с одним целочисленным параметром

Конструктор без параметров имеет вид:

```
PriorityQueue()
```

Он создает очередь с приоритетами начальной емкостью 11, размещающую элементы согласно естественному порядку сортировки, определенному интерфейсом `Comparable`.

Пример.

```
1  import java.util.*;
2  
3  record Person(String firstName,
4                String lastName,
5                int age) implements Comparable<Person>{
6      @Override
7      public int compareTo(Person anotherPerson) {
8          return this.age() - anotherPerson.age();
9      }
10 
11     @Override
12     public String toString() {
13         return "Person{" +
14                 "firstName='" + firstName + '\'' +
15                 ", lastName='" + lastName + '\'' +
16                 ", age=" + age +
```

```
17              '}';
18      }
19  }
20
21  public class CompareDemo
22  {
23      static String toString(PriorityQueue queue) {
24          String str = "";
25          for(var element : queue) {
26              str = str + element.toString() + "\n";
27          }
28          return str;
29      }
30
31      public static void main(String[] args) {
32          PriorityQueue queue = new PriorityQueue();
33          queue.add(new Person("Саша", "Иванов", 36));
34          queue.add(new Person("Маша", "Петрова", 23));
35          queue.add(new Person("Даша", "Сидорова",34));
36          queue.add(new Person("Вася", "Иванов", 25));
37          System.out.println(toString(queue));
38      }
39  }
```

Результат работы программы:

```
Person{firstName='Маша', lastName='Петрова', age=23}
Person{firstName='Вася', lastName='Иванов', age=25}
Person{firstName='Даша', lastName='Сидорова', age=34}
Person{firstName='Саша', lastName='Иванов', age=36}
```

<u>Замечание.</u>

Если из предыдущего кода удалить упоминание о `Comparable` *и переопределение метода* `compareTo()`, *то на этапе выполнения программы возникнет исключительная ситуация в связи с тем, что Java-машина не будет знать как сравнивать пользовательские объекты типа* `Person`.

Конструктор, имеющий один целочисленный параметр, имеет формат:

```
PriorityQueue(int initialCapacity),
```

где целочисленный параметр `initialCapacity` имеет значение начальной емкости коллекции. Понятие емкости уже обсуждалось неоднократно и в данном случае ***не*** имеет ничего нового.

Конструктор, принимающий в качестве значения предварительно инициализированную коллекцию

Формат конструктора:

```
PriorityQueue(Collection<? extends E> c);
```

Пример.

```
1  import java.util.*;
2  public class QueueDemo
3  {
4      public static void main(String[] args) {
5          Vector v = new Vector();
6          v.add(new Integer(36));
7          v.add(new Integer(23));
8          v.add(new Integer(34));
9          v.add(new Integer(25));
10         PriorityQueue queue = new PriorityQueue(v);
11         System.out.println(queue);
12     }
13 }
```

Результат работы программы:

```
[23, 25, 34, 36]
```

Замечание.

Интерфейс `Comparable` не используется. Но класс `Integer` сам реализует интерфейс `Comparable` и для своих объектов переопределяет метод `compareTo()`. Поэтому в данном случае при сортировке применяется отношения сравнения, заданные для типа `Integer` «по умолчанию».

Конструктор, принимающий в качестве значения компаратор

Формат конструктора:

```
PriorityQueue(int initialCapacity,
                         Comparator<? super E> comparator),
```

где первый параметр `initialCapacity` устанавливает начальную емкость и является необязательным, второй параметр получает на вход ссылку типа `Comparator` проинициализированную объектом класса, имплементирующего (реализующего) интерфейс `Comparator`.

Пример.

```
1   import java.util.*;
2   record Person(String firstName,
3                         String lastName, int age) {
4       @Override
5       public String toString() {
6           return "Person{" +
7                   "firstName='" + firstName + '\'' +
8                   ", lastName='" + lastName + '\'' +
9                   ", age=" + age +
10                  '}';
11      }
12  }
13
14  class PersonComparator implements Comparator<Person>{
15      @Override
16      public int compare(Person o1, Person o2) {
17         return o1.lastName().compareTo(o2.lastName());
18      }
19  }
20
21  public class ComparatorDemo
22  {
23      static String toString(PriorityQueue queue) {
24         String str = "";
25         for(var element : queue) {
26             str = str + element.toString() + "\n";
27         }
28         return str;
29      }
```

```
30
31     public static void main(String[] args) {
32         PersonComparator personCompar =
33                             new PersonComparator();
34         PriorityQueue queue =
35            new PriorityQueue (10, personCompar);
36         queue.add(new Person("Саша", "Иванов", 36));
37         queue.add(new Person("Маша", "Петрова", 23));
38         queue.add(new Person("Даша", "Сидорова",34));
39         queue.add(new Person("Вася", "Иванов", 25));
40         System.out.println(toString(queue));
41     }
42 }
```

Результат работы программы:

```
Person{firstName='Саша', lastName='Иванов', age=36}
Person{firstName='Вася', lastName='Иванов', age=25}
Person{firstName='Даша', lastName='Сидорова', age=34}
Person{firstName='Маша', lastName='Петрова', age=23}
```

Очевидно, также допускается следующий синтаксис в методе `main()`:

```
Comparator personCompar = new PersonComparator();
```

__Замечание.__

Также существует конструктор `PriorityQueue(PriorityQueue<? Extends E> c)`*. Он по сути дела является частным случаем уже рассмотренного конструктора, принимающего в качестве значения* ***__любую__*** *инициализированную коллекцию* `PriorityQueue(Collection<? Extends E> c)`*.*

Кроме того существует еще один конструктор `PriorityQueue(SortedSet<? Extends E> c)`*, получающий в качестве параметра ссылку на интерфейс* `SortedSet`*, который будет рассмотрен позже.*

Использование синтаксиса анонимного класса для создания Compparator-a

Предыдущий пример позволяет продемонстрировать еще одну возможность создания компаратора – это использование синтаксиса анонимного класса.

Изменения в предыдущем примере заключаются в том, что описание метода `compare()` перенесено из отдельного класса в анонимный, инициализирующий ссылку на компаратор.

Пример.

```
import java.util.*;
record Person(String first, String last, int age) {
    @Override
    public String toString() {
        return "Person{" +
                "firstName='" + first + '\'' +
                ", lastName='" + last + '\'' +
                ", age=" + age +
                '}';
    }
}

public class Demo
{
    static String toString(PriorityQueue queue) {
        String str = "";
        for(var element : queue) {
            str = str + element.toString() + "\n";
        }
        return str;
    }

   public static void main(String[] args) {
        //анонимный класс для создания компаратора
        Comparator<Person> personCompar =
                        new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                return o1.last().compareTo(o2.last());
            }
        }; //конец анонимного класса
```

```
32
33          PriorityQueue queue =
34                  new PriorityQueue (10, personCompar);
35          queue.add(new Person("Саша", "Иванов", 36));
36          queue.add(new Person("Маша", "Петрова", 23));
37          queue.add(new Person("Даша", "Сидорова",34));
38          queue.add(new Person("Вася", "Иванов", 25));
39          System.out.println(toString(queue));
40      }
41  }
```

Результат работы такой же, как и в предыдущем случае. Напомним, что при переопределении интерфейса с помощью анонимного класса используется синтаксис, на первый взгляд противоречащий ранее перечисленным правилам, а именно тому, что с помощью интерфейса нельзя создавать объект (ключевое слово `new` нельзя использовать, т. к у интерфейса конструкторов нет).

Таким образом строка из примера выше:

```
Comparator<Person> personCompar =
                                        new Comparator<Person>() {
```

на первый взгляд не соответствует принятым ранее правилам. Однако справа «по умолчанию» с помощью `new` создается безымянный объект анонимного класса имплементирующий указанный интерфейс `Comparator` и далее ссылка на объект этого класса присваивается указателю на имплементируемый интерфейс.

Применение нескольких компараторов при сортировке

Представляет интерес возможность применения нескольких компараторов при сортировке. Интерфейс `Comparator` определяет специальный метод по умолчанию `thenComparing()`, который позволяет использовать цепочки компараторов

Пример.

```
1  import java.util.*;
2  record Person(String firstName,
3                         String lastName, int age) {
```

```
4      @Override
5      public String toString() {
6          return "Person{" +
7                  "firstName='" + firstName + '\'' +
8                  ", lastName='" + lastName + '\'' +
9                  ", age=" + age +
10                 '}';
11     }
12 }
13 
14 class NameComparator implements Comparator<Person> {
15     @Override
16     public int compare(Person a, Person b){
17         return a.lastName().compareTo(b.lastName());
18     }
19 }
20 
21 class AgeComparator implements Comparator<Person>{
22     @Override
23     public int compare(Person a, Person b){
24         if(a.age() > b.age()) return 1;
25         else if(a.age()< b.age()) return -1;
26         else return 0;
27     }
28 }
29 
30 public class ComparatorDemo
31 {
32     static String toString(PriorityQueue queue) {
33         String str = "";
34         for(var element : queue) {
35             str = str + element.toString() + "\n";
36         }
37         return str;
38     }
39 
40    public static void main(String[] args) {
41         Comparator<Person> personCompar =
42                 new NameComparator().thenComparing(new
43                                         AgeComparator());
44         PriorityQueue queue =
45                 new PriorityQueue (10, personCompar);
46         queue.add(new Person("Саша", "Иванов", 36));
47         queue.add(new Person("Маша", "Петрова", 23));
```

```
48          queue.add(new Person("Даша", "Сидорова",34));
49          queue.add(new Person("Вася", "Иванов", 25));
50          System.out.println(toString(queue));
51      }
52  }
```

Результат работы программы:

```
Person{firstName='Вася', lastName='Иванов', age=25}
Person{firstName='Саша', lastName='Иванов', age=36}
Person{firstName='Даша', lastName='Сидорова', age=34}
Person{firstName='Маша', lastName='Петрова', age=23}
```

Использование компараторов без создания, имплементирующих интерфейс Comparator классов

Допускается иной синтаксис метода main() без создания класса (или классов, как в примере выше) реализующих интерфейс Comparator.

В случае если разработчик ***не*** хочет создавать соответствующий класс (для одного критерия сравнения) или классы (для нескольких критериев сравнения), то можно использовать соответственно только метод comparing() (для одного критерия) или два метода интерфейса Compapator последовательно comparing() и thenComparing() (для двух критериев).

В качестве параметра методы comparing() и thenComparing() получают выражения типа:

ИмяКласса::ИмяГеттераПоляСортировки

В качестве примера можно сказать, что для класса Person параметрами для методов comparing() и thenComparing() будут выражения Person::lastName и Person::age соответственно.

Пример.

```
1  import java.util.*;
2  record Person(String firstName,
3                           String lastName, int age) {
4      @Override
5      public String toString() {
```

```
 6            return "Person{" +
 7                    "firstName='" + firstName + '\'' +
 8                    ", lastName='" + lastName + '\'' +
 9                    ", age=" + age +
10                    '}';
11        }
12    }
13
14    public class ComparatorDemo
15    {
16        static String toString(PriorityQueue queue) {
17            String str = "";
18            for(var element : queue) {
19                str = str + element.toString() + "\n";
20            }
21            return str;
22        }
23
24        public static void main(String[] args) {
25            Comparator<Person> personCompar = Comparator
26                              .comparing(Person::lastName)
27                              .thenComparing(Person::age);
28            PriorityQueue queue =
29                new PriorityQueue (10, personCompar);
30            queue.add(new Person("Саша", "Иванов", 36));
31            queue.add(new Person("Маша", "Петрова", 23));
32            queue.add(new Person("Даша", "Сидорова",34));
33            queue.add(new Person("Вася", "Иванов", 25));
34            System.out.println(toString(queue));
35        }
36    }
```

Результат работы программы будет такой же, как и в предыдущем случае.

Использование методов `offer()`, `peek()`, `poll()` интерфейса `Queue`

Как уже отмечалось в дополнение к базовым операциям интерфейса `Collection`, очередь предоставляет дополнительные операции вставки, получения и контроля.

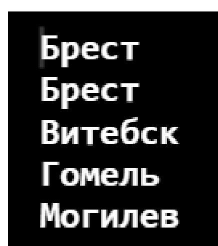
Замечание.

*Очереди обычно, но **не** обязательно, упорядочивают элементы в FIFO (first-in-first-out, "первым вошел - первым вышел") порядке.*

Пример.

```
1   import java.util.*;
2   public class QueueExample
3   {
4       public static void main(String[] args) {
5           Queue queue = new PriorityQueue();
6           queue.add("Витебск");
7           queue.add("Брест");
8           queue.offer("Гомель");
9           queue.offer("Могилев");
10          //вывод на экран элемента приоритетной очереди
11          // без его удаления
12          System.out.println(queue.peek());
13
14          String town;
15          //удаление элемента после вывода на экран
16          while ((town = (String) queue.poll()) != null){
17              System.out.println(town);
18          }
19      }
20  }
```

Результат работы программы:



```
Брест
Брест
Витебск
Гомель
Могилев
```

Метод `offer()` вставляет элемент в очередь, если это не удалось - возвращает `false`. Этот метод отличается от метода `add()` интерфейса `Collection` тем, что метод `add()` может неудачно добавить элемент только с использованием `unchecked` исключения.

Хотя создаваемая очередь `queue` имеет четыре элемента на экране выводятся пять названий городов. Объясняется тем что в середине программы с помощью метода `peek()` осуществляется дополнительный вывод первого элемента очереди.

Приоритетная очередь начинается со слова «Брест», хотя при вводе эта строка не является первой. В этом и состоит особенность приоритетной очереди – в ней храниться элементы, уже отсортированные в смысле естественного порядка, определенного для конкретного типа элементов очереди (тип `String` является `Comparable`-типом).

Метод `poll()` (как и `remove()`) удаляет элемент, расположенный в вершине очереди и возвращают его. Методы `remove()` и `poll()` отличаются лишь поведением, когда очередь пустая: метод `remove()` генерирует исключение, а метод `poll()` возвращает `null`.

К вопросу о создании приоритетной очереди из различных типов данных

Во всех предыдущих коллекциях было продемонстрирована возможность создания коллекции из разных типов данных. Поэтому представляет интерес обсуждение возможности создания приоритетной очереди из разнотипных элементов.

Даже в случае использования различных базовых типов эти значения между собой уже будут несравнимые и поэтому их упорядочить будет невозможно и даже на этапе формирования приоритетной очереди возникнет исключительная ситуация `ClassCastExseption`.

Данную задачу невозможно решить за счет написания своего собственного компаратора. Т.к. по синтаксису компаратор имеет один параметр, определяющий тип и по своей сути может переопределять сравнение только для двух объектов одного типа.

Замечание.

В приоритетной очереди могут храниться только значения одинаковых типов и если необходимо по каким-то причинам хранить значения разных типов, то:

- *они должны быть приведены к одному «стандартному» типу, например к `String` или другому по выбору разработчика;*
- *если «стандартный» тип не является базовым и не является классом `String`, то необходимо создать компаратор для объектов этого «стандартного» типа;*

Однако при этом надо предусмотреть метод декодирования из значения «стандартного» типа в исходный вид.

Рассмотрим простейший пример действий по сохранению любых значений базового типа или класса `String` в очереди:

- в качестве «стандартного» хранения выбран класс `String`;

- создается класс-оболочка для хранения приведенного к одному типу данного;
- при создании объекта класса оболочки его свойство типа строка, получает составное значение: тип хранимого значения указывается в двухсимвольном префиксе перед данным, а данное преобразуется в строковый тип;
- компаратор сравнивает и сортирует два строковых значения из очереди объектов, по результатам сравнения их строковых свойств без учета двух первых символов префикса;
- статический метод `toString()` выводит значения из очереди без двух первых символов префикса;
- декодирование значения из типа `String` в исходный тип осуществляется набором методов в соответствии с хранимым префиксом.

Пример.

```
1  import java.util.*;
2
3  class CastStr<T> {
4      private String str;
5      //два конструктора: один "по умолчанию", второй –
6      //параметризованный
7      CastStr() {};
8      CastStr(T x) {
9          if (x instanceof Integer)
10             this.str = "in" + x.toString();
11         else if (x instanceof Float)
12             this.str = "fl" + x.toString();
13         else if (x instanceof Double)
14             this.str = "do" + x.toString();
15         else if (x instanceof Byte)
16             this.str = "by" + x.toString();
17         else if (x instanceof Boolean)
18             this.str = "bo" + x.toString();
19         else
20             this.str = "st" + x.toString();
21     }
22     //переопределение метода toString() для игнорирования
23     //префиксов
24     @Override
25     public String toString() {
26         return this.str.substring(2);
27     }
28
29     //методы декодирования каждого типа в отдельности
30     Integer decodingInt() {
```

```
31          return Integer.parseInt( this.str.substring(2));
32      }
33      Double decodingDouble() {
34          return Double.parseDouble( this.str.substring(2));
35      }
36      Float decodingFloat() {
37          return Float.parseFloat( this.str.substring(2));
38      }
39      Byte decodingBoolean() {
40          return Byte.parseByte( this.str.substring(2));
41      }
42      Boolean decodingByte() {
43          return Boolean.parseBoolean(this.str.substring(2));
44      }
45      String decodingStr() {
46          return this.str.substring(2);
47      }
48      //геттер для компаратора
49      String get() {
50          return str;
51      }
52  }
53
54  //компаратор
55  class CastStrComparator implements Comparator<CastStr>{
56      @Override
57      public int compare(CastStr obj1, CastStr obj2) {
58          String obj1Str = obj1.get().substring(2);
59          String obj2Str = obj2.get().substring(2);
60          return obj1Str.compareTo(obj2Str);
61      }
62  }
63
64  public class Example
65  {
66      public static void main(String[] args) {
67          //создание ссылки на компаратор
68          CastStrComparator strCompar =
69                                  new CastStrComparator();
70          //объявление очереди, использ. пользов. компаратор
71          PriorityQueue queue = new PriorityQueue(strCompar);
72          //заполняем очередь разнотипными данными с промеж.
73          //приведением их к "стандартному типу" String
74          queue.add(new CastStr("Витебск"));
75          queue.add(new CastStr(257));
76          queue.offer(new CastStr(3.14));
77          queue.offer(new CastStr(true));
78          //поиск в очереди значений определенного типа
79          //Integer и String
80          CastStr element = new CastStr();
```

```
81          while ((element = (CastStr) queue.poll()) != null) {
82              //prefix удерживает два первых символа из объекта
83              //очереди для декодирования значения
84              String prefix = element.get().substring(0,2);
85              switch(prefix){
86                  case "in": {
87                      //блок нужен для var temp по каждой ветке
88                      var temp = element.decodingInt();
89                      System.out.print(temp);
90                      System.out.println("\t\t -Это значение "+
91                                      "типа Integer? " +
92                                      (temp instanceof Integer));
93                      break;
94                  }
95                  case "st": {
96                      //из-за блока temp уже здесь не видна
97                      var temp = element.decodingStr();
98                      System.out.print(temp);
99                      System.out.println("\t -Это значение "+
100                                     "типа String? " +
101                                     (temp instanceof String));
102                     break;
103                 }
104             }
105         }
106     }
107 }
```

Результат работы программы:

```
257       -Это значение типа Integer? true
Витебск   -Это значение типа String? true
```

Применение приоритетной очереди для сортировки элементов

Одним из вариантов использования приоритетной очереди является автоматическая сортировка набора элементов других коллекций.

Замечание.

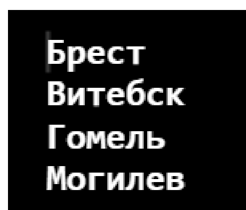
Однако в этом случае коллекция, в которой будет осуществляться сортировка должна содержать элементы одного типа, причем два данных этого тип должны быть `Comparable` *(сравнимы) иначе придется писать еще и дополнительный компаратор.*

Пример.

```
import java.util.*;
public class QueueExample
{
    static PriorityQueue heapSort(Collection c) {
       return new PriorityQueue(c);
    }

    public static void main(String[] args) {
       Vector v = new Vector();
       v.add("Витебск");
       v.add("Брест");
       v.add("Гомель");
       v.add("Могилев");
       v = new Vector(heapSort(v));
       for(int i = 0; i < v.size(); i++){
          System.out.println(v.get(i));
       }
    }
}
```

Результат работы программы:



Перечислим собственно методы коллекции `PriorityQueue`

Кроме реализации методов интерфейса `Queue` данная коллекция имеет еще некоторые дополнительные экземплярные методы:

- `void clear()` - удаляет все элементы из очереди;
- `Comparator<? Super E> comparator()` - возвращает компаратор, используемый для упорядочения элементов в этой очереди, или ноль, если эта очередь отсортирована в соответствии с естественным порядком ее элементов;
- `boolean contains(Object obj)` - возвращает `true`, если эта очередь содержит указанный элемент;

- Iterator<E> iterator() - возвращает итератор по элементам в очереди;
- int size() - возвращает количество элементов в этой коллекции;
- Object[] toArray() - возвращает массив, содержащий все элементы в этой очереди;
- <T> T[] toArray(T[] a) - возвращает массив, содержащий все элементы этой очереди; тип возвращаемого массива во время выполнения - это тип указанного массива.

Интерфейс Deque

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).

Методы интерфейса

Интерфейс имеет следующие основные методы:

- 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;
- void **clear**() - удаляет все элементы из двусвязной (двусторонней) очереди;
- ArrayDeque<E> **clone**() – возвращает копию двусвязной (двусторонней) очереди;
- 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;

- 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;
- 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;
- 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` — реализация интерфейса `Deque`, который расширяет интерфейс `Queue` методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out).

Коллекция `ArrayDeque` представляет собой реализацию с использованием массивов, подобно `ArrayList`.

Особенности:

- не позволяет обращаться к элементам по индексу;
- запрещено хранение значения `null`;
- не поддерживает многопоточный безопасный доступ;
- работает быстрее чем `Stack`, если используется как LIFO коллекция, а также быстрее чем `LinkedList`, если используется как FIFO;
- позволяет работать со списком с обоих концов;
- некоторые методы могут выбрасывать исключения;
- поддержка двунаправленного обхода итератора.

В классе `ArrayDeque` определены следующие конструкторы:

- `ArrayDeque()` - этот конструктор используется для создания пустого `ArrayDeque` и по умолчанию имеет начальную емкость для хранения 16 элементов.

  Пример.
  ```
  ArrayDeque<ТипЗначения> dq =
                          new ArrayDeque();
  ```

- `ArrayDeque(Collection<? extends E> c)` - создает очередь, наполненную элементами из коллекции `c`.

  Пример.
  ```
  ArrayDeque<ТипЗначения> dq =
           new ArrayDeque<E>(Collection c);
  ```

- `ArrayDeque(int capacity)` - создает очередь с начальной емкостью `capacity`. Если мы явно не указываем начальную емкость, то емкость по умолчанию будет равна 16.

Пример.

```
ArrayDeque<ТипЗначения> dq =
                new ArrayDeque(capacity);
```

Использования методов класса ArrayDeque

Рассмотрим ряд примеров использования методов коллекции `ArrayDeque`.

Пример.

```
1   import java.util.*;
2
3   record Person(String name) {}
4
5   public class ExamplesArrayDeque
6   {
7       public static void main(String[] args) {
8           ArrayDeque arrDeque = new ArrayDeque();
9           //стандартное добавление элементов
10          arrDeque.add("Германия");
11          arrDeque.addFirst(1); //добавляем элемент в начало
12          arrDeque.push(3.14); // добавляем элемент в начало
13          arrDeque.addLast('И');//добавляем элемент в конец
14          arrDeque.add(new Person("Ярополк"));
15          // получаем первый элемент без удаления
16          var sFirst = arrDeque.getFirst();
17          System.out.println("Первый элемент: " + sFirst);
18          // получаем последний элемент без удаления
19          var sLast = arrDeque.getLast();
20          System.out.println("Последний элемент: " +sLast);
21          //используем метод printf
22          System.out.printf("Queue size: %d \n",
23                            arrDeque.size());
24          // перебор коллекции
25          System.out.println("\nКоллекция arrDeque:");
26          while(arrDeque.peek()!=null){
27              //извлечение с удалением из начала
28              //после цикла очередь arrDeque будет пуста
29              System.out.println(arrDeque.pop());
30          }
```

```
31        // очередь только из объектов Person
32        ArrayDeque<Person> people =
33                                  new ArrayDeque<Person>();
34        people.addFirst(new Person("Антон"));
35        people.addLast(new Person("Сергей"));
36        System.out.println("\nКоллекция people:");
37        // перебор без извлечения
38        for(Person p : people){
39            System.out.println(p.name());
40        }
41        //очистка очереди people
42        people.clear();
43        System.out.println("\nКоллекция people " +
44                           "после clear():\n" + people);
45        System.out.println("Коллекция people пуста? " +
46                           people.isEmpty());
47    }
48 }
```

Результат работы программы:

```
Первый элемент: 3.14
Последний элемент: Person[name=Ярополк]
Queue size: 5

Коллекция arrDeque:
3.14
1
Германия
И
Person[name=Ярополк]

Коллекция people:
Антон
Сергей

Коллекция people после clear():
[]
Коллекция people пуста? true
```

Замечание.

Хотя в объекте коллекции `ArrayDeque` можно хранить разнотипные элементы, но ***обычно*** *очереди и стеки применяются для хранения однотипных элементов.*

Следующий пример в основном посвящен методам не использованным в предыдущем примере.

Пример.

```
1  import java.util.*;
2  public class DemoArrayDeque
3  {
4      public static void main(String[] args) {
5          Deque<Integer> deQue = new ArrayDeque<>(10);
6          // методы addFirst() и addLast
7          deQue.addFirst(564);
8          deQue.addFirst(291);
9          deQue.addLast(24);
10         deQue.addLast(14);
11         // использование итераторов
12         System.out.println("Вывод на экран очереди" +
13                          "с использованием итераторов:");
14         System.out.println("- прямой порядок:");
15         for (Iterator itr = deQue.iterator();
16             itr.hasNext();
17             /*третья секция for()   пусто*/) {
18             System.out.print(itr.next() + " ");
19         }
20         // descendingIterator() для обратного порядка
21         System.out.println("\n- обратный порядок:");
22         for (Iterator dItr = deQue.descendingIterator();
23             dItr.hasNext();
24             /*третья секция for()   пусто*/) {
25              System.out.print(dItr.next() + " ");
26         }
27         //Вывод на экран головного элемента списка (начало)
28         System.out.println("\n\nelement(). Значение из " +
29                            "вершины списка : " +
30                            deQue.element());
31         System.out.println("Список не изменился: " +
32                            deQue);
33         System.out.println("\npeek(). Значение из " +
34                            "вершины списка : " +
35                            deQue.peek());
36         System.out.println("Список не изменился: " +
37                            deQue);
38         System.out.println("\npoll(). Значение из " +
39                            "вершины списка : " +
40                            deQue.poll());
41         System.out.println("У списка удален " +
42                            "головной элемент: " + deQue);
```

```
43          System.out.println("\nremove(). Значение из " +
44                              "вершины списка : " +
45                              deQue.remove());
46          System.out.println("У списка удален " +
47                              "головной элемент: " + deQue);
48          // toArray() method :
49          Object[] arr = deQue.toArray();
50          System.out.println("\nArray Size : " +
51                              arr.length);
52          System.out.println(Arrays.toString(arr));
53      }
54  }
```

Результат работы программы:

```
Вывод на экран очередис использованием итераторов:
- прямой порядок:
291 564 24 14
- обратный порядок:
14 24 564 291

element(). Значение из вершины списка : 291
Список не изменился: [291, 564, 24, 14]

peek(). Значение из вершины списка : 291
Список не изменился: [291, 564, 24, 14]

poll(). Значение из вершины списка : 291
У списка удален головной элемент: [564, 24, 14]

remove(). Значение из вершины списка : 564
У списка удален головной элемент: [24, 14]

Array Size : 2
[24, 14]
```

Примеры статических и экземплярных generic-методов для работы с объектом коллекции ArrayDeque

Приведем примеры generic-методов для работы с коллекцией `ArrayDeque`.

Пример.

```
1  import java.util.*;
2  public class Program {
3      //статический метод
4      static ArrayDeque generArrDeq(final int MIN,
5                                    final int MAX,
6                                    int n) {
7          ArrayDeque arrDeq = new ArrayDeque(n);
8          //генерация случайного индекса для вставки строки в
9          //середину коллекции (не нулевого и не последнего)
10         int m =
11         (new Double(Math.random() * (n - 2) + 1)).intValue()
12         //заполнение коллекции целыми числами до
13         //сгенерированного индекса
14         for(int i = 0; i < m; i++) {
15             arrDeq.add((new Double(Math.random() *
16                             (MAX - MIN) + MIN)).intValue());
17         }
18         //вставка в коллекцию строки
19         arrDeq.addLast("Строка");
20         //заполнение остатка коллекции числами типа Double
21         for(int i = m + 1; i < n; i++) {
22             arrDeq.add(new Double(Math.random() *
23                                 (MAX - MIN) + MIN));
24         }
25         return arrDeq;
26     }
27     //экземплярный метод сдвига влево
28     void shiftLeft(ArrayDeque a) {
29         var temp = a.removeFirst();
30         a.addLast(temp);
31     }
32     //экземплярный метод сдвига вправо
33     void shiftRight(ArrayDeque a) {
34         var temp = a.removeLast();
35         a.addFirst(temp);
36     }
37     //статический метод реверса коллекции
38     static void reverse(ArrayDeque a) {
39         //создаем копию коллекции
40         ArrayDeque copy = a.clone();
41         //очищаем исходную коллекцию
42         a.clear();
43         //собственно реверс при копировании из копии в
44         //исходную коллекцию
45         for(Iterator dItr = copy.descendingIterator();
46             dItr.hasNext();
47             /*третья секция for()   пусто*/) {
```

```
48 |            a.add(dItr.next());
49 |        }
50 |    }
51 |
52 |    public static void main (String args[]) {
53 |        final int MIN = 0;  //нижняя граница генерации
54 |        final int MAX = 10; //верхняя граница генерации
55 |        ArrayDeque a = generArrDeq(MIN, MAX, 5);
56 |        System.out.println("Исходная коллекция");
57 |        System.out.println(a.toString());
58 |        Program obj = new Program();
59 |        System.out.println("Циклический сдвиг влево");
60 |        obj.shiftLeft(a);
61 |        System.out.println(a.toString());
62 |        System.out.println("Циклический сдвиг вправо");
63 |        obj.shiftRight(a);
64 |        System.out.println(a.toString());
65 |        System.out.println("Реверс");
66 |        reverse(a);
67 |        System.out.println(a.toString());
68 |    }
69 | }
```

Результат работы программы:

```
Исходная коллекция
[3, 8, Строка, 0.04992279465436345, 1.025432172231503]
Циклический сдвиг влево
[8, Строка, 0.04992279465436345, 1.025432172231503, 3]
Циклический сдвиг вправо
[3, 8, Строка, 0.04992279465436345, 1.025432172231503]
Реверс
[1.025432172231503, 0.04992279465436345, Строка, 8, 3]
```

Коллекция LinkedList

`LinkedList` — коллекция, реализующая интерфейсы `Dequeue`, `Queue` и `List`.

Позволяет хранить любые данные, включая `null`. Особенностью реализации данной коллекции является то, что в ее основе лежит двунаправленный связный список (каждый элемент имеет ссылку на

предыдущий и следующий). Так же, ввиду особенностей реализации, данную коллекцию можно использовать как стек или *очередь*.

Свойства коллекции:

- можно организовать стек (`LIFO`), очередь (`FIFO`), или двустороннюю очередь (очередь с возможностью добавления в начало или конец), со временем доступа `O(1)`;
- 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`. Для хранения примитивных типов использует соответствующие классы-обертки;
- не синхронизирована (т.е. при многопоточном программировании потоко***не***безопасена).

В классе `LinkedList` определены следующие конструкторы:

- `LinkedList()` - этот конструктор используется для создания пустого списка.

Пример.

```
LinkedList<ТипЗначения> dq =
                     new LinkedList();
```

- `LinkedList(Collection<? extends E> c)` - создает очередь, наполненную элементами из коллекции `c`.

Пример.

```
LinkedList<ТипЗначения> dq =
       new LinkedList(Collection c);
```

Пример.

```
1   import java.util.*;
2   public class Countdown {
3       public static void main(String[] args) {
4           int time = 10;
5           Queue<Integer> queue = new LinkedList<>();
6           //заполнение очереди целыми значениями в обратном
7           //порядке
8           for (int i = time; i >= 0; i--) {
9               queue.add(i);
10          }
```

```
11            while (!queue.isEmpty()) {
12                System.out.print(queue.remove() + " ");
13            }
14        }
15    }
```

Результат работы программы:

```
10 9 8 7 6 5 4 3 2 1 0
```

Коллекция `LinkedList` не имеет методов отличных от методов, определенных интерфейсами `Deque` и `List`.

Фактическая реализация этих методов для класса `LinkedList` полностью повторяет реализацию для уже рассмотренной коллекции `ArrayDeque` (интерфейс `Deque`), а также `ArrayList` (интерфейс `List`).

Поэтому все примеры по использованию `LinkedList` с точностью до замены типа могут быть взяты из раздела посвященного коллекциям `ArrayDeque`, а также `ArrayList`.

Коллекции, реализующие Deque vs коллекции, реализующие только List

Прежде всего, в работе с серединой списка. Вставка в середину и удаление из середины коллекций, реализующих `Deque` устроены гораздо проще, чем в коллекциях, реализующих только интерфейс `List`. Это объясняется тем, что при использовании реализаций методов интерфейса `Deque` можно просто переопределить ссылки соседних элементов, а ненужный элемент исключить из цепочки ссылок или наоборот нужный вставить в цепочку.

В то время как в коллекциях, реализующих только интерфейс `List` при операциях вставки и удаления необходимо:

- при вставке:
  - проверить, хватает ли места (`capacity`) для хранения увеличенного массива;
  - если не хватает `capacity`, то создать новый массив и скопировать туда данные;
- собственно удаление/вставка элемента, осуществляется сдвигом всех остальных элементов вправо/влево (в зависимости от типа операции). Причем длительность этого

процесса сильно зависит от количества хранимых элементов. Одно дело сдвинуть десять элементов, и совсем другое - миллион.

То есть, если в разрабатываемой программе чаще происходят операции вставки/удаления в/из середины списка, то коллекции, реализующие `Deque`, должны работать быстрее, чем коллекции, реализующие только `List`.

Однако если учитывать время, затрачиваемое на обязательную предварительную операцию поиска удаляемого или места вставляемого элемента в коллекциях реализующих `Deque` (это использование итератора за время `O(n)`) и в коллекции реализующей `List` (это использование метода `get()` за время `O(1)`), то говорить о явных преимуществах одной коллекции перед другой невозможно.

Дополнительные примеры методов для работы с коллекциями. Специфика создания `generic`-методов для обработки коллекций

Приведем примеры действий, которые еще не рассматривались. В частности, пусть перед разработчиком стоит задача написать generic-методы, работающие только с числовыми данными. Имеется ввиду методы, обрабатывающие не только один тип данных в коллекции, жестко заданный еще на этапе его инициализации, а позволяющий обрабатывать произвольный набор числовых данных ***без промежуточного приведения*** их типа к единому (например, `Double`).

Под обработкой в данном разделе понимается не манипуляции с изменение порядка значений, а обычные вычислительные действия:

- изменение знака отрицательного числа на противоположный;
- прибавление какой-либо константы к некоторым значениям коллекции, при этом должен сохраниться исходный тип данного.

В этой ситуации, казалось бы, верным решением будет использовать явную параметризацию коллекции с хорошо известным выражением `<T extends Numbers>`. Но, как ни странно, для Java, это будет неверным выбором. В подавляющем большинстве случаев (все зависит от алгоритма решаемой задачи) компилятор не пропустит подобную параметризацию методов из-за того, что в теле методов будут присутствовать выражения для преобразования типа `Object` к параметризованному типу `T`. Выполнить же явное приведение типов в этом случае нет никакой возможности.

Однако если использовать в качестве явного значения типа суперкласс Number, то все радиально поменяется.

С одной стороны, это уже не generic-метод, т.к. явно указан тип, но поскольку это суперкласс, то все коллекции с этим суперклассом допускают их использование для хранения значений имеющих типы наследников т.е. всех числовых типов.

Использование в качестве типа абстрактного класса Number приводит к тому, что для этого типа естественно ни одна арифметическая или логическая операции не определены и зачастую придется (в зависимости от алгоритма) в создаваемых методах явно работать с каждым классом наследником Number в отдельности.

Замечание.

В примерах используется коллекция LinkedList, т.к. она рассматривалась последней, но технологические и методологические проблемы, решаемые в примерах характерны для любой из рассмотренных ранее коллекций, что можно установить прямым изменением типа коллекции и соответствующей заменой методов.

Пример.

```
1  import java.util.*;
2  public class Program {
3      //экземплярный метод определения количества положит.
4      int numberPosit(LinkedList<Number> a) {
5          int n = 0;
6          for(var element : a) {
7          //универсальным типом для сравнения с нулем
8          //в Jave является double, т.к. имеет самую малую
9          //величину хранимого дробного числа
10         //Double.MIN_VALUE, определенного в классе-обертке
11         //Double с практической точки зрения все, что
12         //меньше этой константы можно считать нулем
13             if (element.doubleValue() > Double.MIN_VALUE) {
14                 n++;
15             }
16         }
17         return n;
18     }
19     //экземплярный метод конвертации коллекции, из
20     //отрицательных и положительных числовых значений в
21     //коллекцию только из положительных чисел
22     void toPositive(LinkedList<Number> a) {
23         //создаем копию коллекции, пришедшую на вход
24         LinkedList copy = (LinkedList) a.clone();
25         //очищаем исходную коллекцию
26         a.clear();
```

```
26          a.clear();
27          //перебираем значения копии коллекции итератором
28          Iterator itr = copy.iterator();
29          while(itr.hasNext()) {
30              //temp   текущее значение коллекции
31              var temp = (Number) itr.next();
32              //определяем отрицательное ли это значение
33              if (temp.doubleValue() < -Double.MIN_VALUE) {
34                  //для сохранения типа значения и изменения
35                  //только его знака выполняем анализ типа
36                  //данного из коллекции и производим
37                  //преобразование отрицательного значения в
38                  //положительное
39                  if (temp instanceof Double) {
40                      temp = new Double(-temp.doubleValue());
41                  }
42                  else if (temp instanceof Float) {
43                      temp = new Float(-temp.floatValue());
44                  }
45                  else if (temp instanceof Integer) {
46                      temp = new Integer(-temp.intValue());
47                  }
48                  else if (temp instanceof Long) {
49                      temp = new Long(-temp.longValue());
50                  }
51                  else {
52                      //исключение: тип данного не определен
53                      try {
54                          throw new Exception("тип данного " +
55                                              "не определен");
56                      }
57                      catch(Exception ex) {
58                          System.out.println(ex.getMessage());
59                      }
60                  } //конец else
61              }
62              //вносим значение того же типа, но если
63              //потребовалось с измененным знаком в коллекцию
64              //пришедшую на вход метода
65              a.add(temp);
66          }
67      }
68      //контроль типов значений в коллекции, которые могли
69      //измениться после преобразования знаков
70      static void typesCollection(LinkedList<Number> a) {
71          Iterator itr = a.iterator();
72          while(itr.hasNext()) {
```

```
73              System.out.println(itr.next().getClass());
74          }
75      }
76
77      public static void main (String args[]) {
78          LinkedList a =
79              new LinkedList(Arrays.asList(-1f, -2, 0L, 4f));
80          System.out.println("Исходная коллекция");
81          System.out.println(a.toString());
82          Program obj = new Program();
83          System.out.println("Число положительных: " +
84                                  obj.numberPosit(a));
85          obj.toPositive(a);
86          System.out.println(a);
87          System.out.println("Контрольный список типов: ");
88          typesCollection(a);
89      }
90  }
```

Результат работы программы:

```
Исходная коллекция
[-1.0, -2, 0, 4.0]
Число положительных: 1
[1.0, 2, 0, 4.0]
Контрольный список типов:
class java.lang.Float
class java.lang.Integer
class java.lang.Long
class java.lang.Float
```

Таким образом из результатов работы программы видно, что типы в коллекции после преобразования знаков не изменились.

Замечание.

Недостатком подобного решения является огромная лестница `if-else-if` *для перебора конкретных типов значений коллекции.*

Необходимость ее присутствия диктуется тем, что:

- *арифметические и логические операции в Java определены только для конкретных типов и их нельзя перегрузить или каким-либо образом переопределить;*
- *она позволяет сгенерировать исключительную ситуацию в случае, если у одного из элементов коллекции будет тип* `Short` *или* `Byte`*.*

Однако эта лестница выглядит крайне непрофессионально и ее следовало бы вынести в отдельный метод.

Теперь несколько изменим задачу: в коллекции **LinkedList** каждому стоящему на четных индексах *числовому* значению (типа класса-обертки) прибавим число **100**. Тип значения необходимо сохранить, однако предполагаем, что значения типа Short или Byte также должны генерировать исключения.

В данном примере вынесем в отдельный метод прибавление 100 к различным типам данных (метод статический метод add100()), а выполнение этой операции для четных индексов будем осуществлять в другом методе (метод editData()) с помощью метода-признака необходимости преобразований (evenIndex()).

Замечание.

Коллекция LinkedList индексации не имеет, поэтому для определения условного индекса будет использоваться вспомогательная целочисленная переменная.

Пример.

```
1   import java.util.*;
2   public class Program
3   {
4       //метод «что делаем» по арифметике;
5       //если необходимо, то такое же пишется для сравнений с
6       //каким-либо значением и прочее
7       static Number add100(Number temp) {
8           if (temp instanceof Double) {
9               return new Double(temp.doubleValue() + 100);
10          }
11          else if (temp instanceof Float) {
12              return new Float(temp.floatValue() + 100);
13          }
14          else if (temp instanceof Integer) {
15              return new Integer(temp.intValue() + 100);
16          }
17          else if (temp instanceof Long) {
18              return new Long(temp.longValue() + 100);
19          }
20          //исключение: тип данного не определен
21          try {
22              throw new Exception("тип не определен");
23          }
```

```
24         catch(Exception ex) {
25             System.out.println(ex.getMessage());
26         }
27         return null;
28     }
29     //метод «для каких данных делаем»
30     static boolean evenIndex(int i) {
31         return i % 2 == 0;
32     }
33     //метод перебора
34     static void editData(LinkedList<Number> a) {
35         LinkedList copy = (LinkedList) a.clone();
36         a.clear();
37         Iterator itr = copy.iterator();
38         int i = 0; //вспомогательная перем. для индекса
39         while(itr.hasNext()) {
40             var temp = (Number) itr.next();
41             if (evenIndex(i)) {
42                 temp = add100(temp);
43             }
44             a.add(temp);
45             i++;
46         }
47     }
48 
49     public static void main (String args[]) {
50         LinkedList a =
51           new LinkedList(Arrays.asList(-1f, -2, 0L, 4f));
52         System.out.println("Исходная коллекция");
53         System.out.println(a.toString());
54         editData(a);
55         System.out.println("Результат");
56         System.out.println(a);
57     }
58 }
```

Результат работы программы:

```
Исходная коллекция
[-1.0, -2, 0, 4.0]
Результат
[99.0, -2, 100, 4.0]
```

С учетом того, что Java тяготеет к использованию *<u>однотипных данных в коллекциях</u>*, то представлял бы интерес пример `generic`-метода, который будет приводить типы уже целой коллекции, а не отдельных

значений в ней. Но это сделать невозможно Java машина не различает `LinkedList<Integer>` и, например, `LinkedList<Double>`. Это для нее одно и тоже (а конкретно `LinkedList<E>`).

Таким образом выполнить приведение типов «руками» с помощью перегрузки нескольких методов не получиться.

Но, как ни странно, с другой стороны, например, тип `LinkedList<Integer>` (как собственно с любым другим конкретным классом-оберткой числовых типов) не приводится к типу `LinkedList<Number>`.

Из этого следует, что и параметризацию методов использовать нельзя для приведения типов. Именно эта последняя особенность не позволяет вернуть коллекцию, состоящую из значений одного *конкретного* класса-обертки через ссылку на объект, как казалось, обещающего типа `LinkedList<Number>`.

Общие сведения о хэш-таблицах

Хеш-таблица (`hash table`) — это специальная структура данных для хранения пар ключей и их значений. По сути, это ассоциативный массив, в котором ключ обрабатывается хеш-функцией.

Пожалуй, главное свойство хеш-таблиц — все три операции: вставка, поиск и удаление — в среднем выполняются за время `O(1)`, среднее время поиска по ней также равно `O(1)` и `O(n)` в худшем случае.

Для того, чтобы разобраться, что такое хеш-таблицы, необходимо представить, что стоит задача создать библиотеку и заполнить ее книгами. Естественно, в этом случае необходимо придумать порядок заполнения.

Первое, что приходит в голову — разместить все книги в алфавитном порядке и записать все в некий алфавитный справочник. В этом случае не придется искать нужную книгу по всей библиотеке, а только по справочнику, в котором будет указан шкаф и полка, на которой храниться книга. Этот справочник, выдающий по автору или названию, адрес хранения и есть хэш-функция.

Хеш-функция имеет следующие свойства:

- всегда возвращает один и тот же адрес для одного и того же ключа;
- не обязательно возвращает разные адреса для разных ключей (коллизии, они же столкновения);
- использует все адресное пространство с одинаковой вероятностью;
- быстро вычисляет адрес расположения элемента.

Поэтому в хэш-таблицах актуальным является разработка методов, предотвращающих коллизии (столкновения). Но это не входит в данный учебный курс.

Таким образом, хеширование и хеш-таблицы применяются для более удобного хранения пар «ключ-значение». Это контейнер используют, если хотят быстро выполнять операции вставки/удаления/нахождения.

Интерфейс Map

Интерфейс `Map<K, V>` представляет соответствие или иначе говоря словарь, где каждый элемент представляет пару «ключ-значение». При этом все ключи должны быть уникальными в рамках объекта `Map`. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

Следует отметить, что в отличие от других интерфейсов, которые представляют коллекции, интерфейс `Map` ***НЕ*** расширяет интерфейс `Collection`.

Среди методов интерфейса `Map` можно выделить следующие:

- `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`**`()` - метод получения объекта в виде коллекции `Set`;
- `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` **`getOrDefault`**`(Object k, V defaultValue)` - возвращает значение объекта, ключ которого равен `k`. Если такого элемента не окажется, то возвращается значение `defaultValue`;
- `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;

- V **putIfAbsent**(K k, V v) - помещает в коллекцию новый объект с ключом k и значением v, если в коллекции еще нет элемента с подобным ключом;
- Set<K> **keySet**() - возвращает набор всех ключей соответствия;
- Collection<V> **values**() - возвращает набор всех значений соответствия;
- void **putAll**(Map<? extends K, ? extends V> map) - добавляет в коллекцию все объекты из соответствия map;
- V **remove**(Object k) - удаляет объект с ключом k;
- int **size**() - возвращает количество элементов коллекции.

Для того чтобы добавить объект в коллекцию, используется метод put(), а чтобы получить по ключу - метод get(). Реализация интерфейса Map также позволяет получить наборы как ключей, так и значений.

Интерфейс Map имеет ***вложенный*** интерфейс Map.Entry<K, V> также как и Map имеет два параметра с ключ типа K и значение типа V. Вложенный интерфейс позволяет определить следующие методы:

- boolean **equals**(Object obj) - возвращает true, если объект obj, передаваемый по ссылке типа Map.Entry, идентичен текущему;
- K **getKey**() - возвращает ключ по ссылке Map.Entry на объект коллекции;
- V **getValue**() - возвращает значение по ссылке Map.Entry на объект коллекции;
- V **setValue**(V v) - устанавливает для текущего объекта значение v;
- int **hashCode**() - возвращает хеш-код данного объекта Map.Entry.

При переборе объектов коллекции будем оперировать этими методами для работы с ключами и значениями объектов.

Коллекция Hashtable

Коллекция Hashtable — реализация такой структуры данных, как хэш-таблица. Она не позволяет использовать null в качестве значения или ключа. Эта коллекция была реализована раньше, чем

Java Collection Framework, но в последствии была включена в его состав.

Как и другие коллекции, созданные на заре Java, Hashtable является синхронизированной (почти все методы помечены как synchronized). Из-за этой особенности у неё имеются существенные проблемы с производительностью и в большинстве случаев рекомендуется использовать другие реализации интерфейса Map ввиду отсутствия у них синхронизации.

Замечание.

Под синхронизированностью (synchonized) имеется ввиду следующее: только один поток может изменить одну таблицу в одно и то же время. Когда какой-то поток работает с объектом Hashtable, то он закрыт для других потоков, пока не отработает первый.

Справедливо утверждение: если синхронизировано, то потокобезопасно.

Коллекция Hashtable реализует интерфейс Map, которая хранит пары «ключ-значения». Hashtable не допускается null-ключей и дублирующих ключей, а также null-значений.

Замечание.

Порядок итерирования (обхода) коллекции совпадает с порядном добавления элементов.

У экземпляра HashMap есть два параметра, которые влияют на его производительность: начальная емкость (capacity) и коэффициент загрузки (loadLactor). Коэффициент загрузки указывает при каком уровне заполнения от установленной емкости происходит ее увеличение (по некотором источникам – почти удвоение). loadLactor – коэффициет позволяющий оптимизировать время необходимое для выполнения операций. Значение этого коэффициента равное 0.75 считается оптимальным.

Конструкторы Hashtable:

- Hashtable() - создание хеш-таблицы с емкостью по умолчанию (11 записей) и коэффициент загрузки по умолчанию (0.75).

Пример.

```
Hashtable<ТипКлюча, ТипЗначения> ht =
                        new Hashtable();
```

- Hashtable(int initialCapacity) - создание хеш-таблицы с указанной начальной емкостью и коэффициентом загрузки по умолчанию (0.75).

  Пример.
  Hashtable~~<ТипКлюча, ТипЗначения>~~ ht =
  new **Hashtable**(20);

- Hashtable(int initialCapacity, float loadFactor)- создание хеш-таблицы с указанной начальной емкостью и указанным коэффициентом загрузки.

  Пример.
  Hashtable~~<ТипКлюча, ТипЗначения>~~ ht =
  new Hashtable(20, 0.9);

- Hashtable(Map<? Extends K, ? extends V> t) - создание хеш-таблицы, инициализированной другой хэш-таблицей.

  Пример.
  Hashtable~~<ТипКлюча, ТипЗначения>~~ ht =
  new Hashtable();
  Hashtable~~<ТипКлюча, ТипЗначения>~~ htNew =
  new Hashtable(ht);

Параметры ~~<ТипКлюча, ТипЗначения>~~ вообще говоря не обязательны. Даже при использовании методов устаревшего интерфейса Enumerator указывать конкретные типы ~~<ТипКлюча, ТипЗначения>~~ при создании HashMap ***не***обязательно.

Однако если пользоваться методами вложенного интерфейса Map.Entry, то явное указание типов обязательно. Иначе невозможно будет обойти коллекцию.

Замечание.

При использовании интерфейса Map.Entry из-за необходимости перечисления конкретных типов создать generic-методы для обработки этой коллекции нельзя.

Как и всегда при создании объекта HashMap ссылку на него можно присваивать ссылке на интерфейс Map, но опять же с конкретным указание типов:

~~Map<ТипКлюча, ТипЗначения>~~ Имя = new **Hashtable**();

Замечание.

Необходимость указания конкретных типов, как и раньше определяется использованием методов интерфейса `Map.Entry`.

Дополнительные методы Hashtable

`Hashtable` реализует все методы интерфейса `Map` и включает дополнительно следующие:

- `Object` **`clone`**`()` - создание копии хеш-таблицы;
- `boolean` **`contains`**`(Object value)` - метод проверки присутствия объекта `value` в хеш-таблице;
- `Enumeration<V>` **`elements`**`()` - получение ключей хеш-таблицы в виде объекта `Enumeration`;
- `Enumeration<K>` **`keys`**`()` - метод получения ключей хеш-таблицы в виде объекта `Enumeration`;
- `void` **`rehash`**`()` - реорганизация `hash`-ключей и увеличение емкости.

Замечание.

Напомним, что интерфейс `Enumeration` *считается устаревшим, поэтому использовать методы* `elements()` *и* `keys()` *в новом коде не рекомендуется.*

В качестве примера рассмотрим задачу организации небольшого телефонного справочника с помощью `Hashtable`. Ключом в примере будет являться `имя+перваяБукваФамилии`.

С помощью метода `put()` программа добавляет новые записи в хеш-таблицу.

Пример.

```
1  import java.util.*;
2  public class Program
3  {
4      //статический не generic метод, т.к. указана типизация
5      //ключей и значений - особенность использ. Map.Entry
6      static void display(Hashtable<String, String> ht){
```

```
 7         for(var item : ht.entrySet()) {
 8             String user  = (String) item.getKey();
 9             String value = (String) item.getValue();
10             System.out.println("Имя (key) = " + user +
11                                 ", Телефон (value) = " + value);
12         }
13     }
14 
15     public static void main (String args[]) {
16         Hashtable<String, String> phoneBook =
17                                       new Hashtable();
18         phoneBook.put("Оксана Л.", "(926) 111-222-333");
19         phoneBook.put("Сергей М.", "(929) 333-222-111");
20         phoneBook.put("Петр Ж."  , "(915) 333-111-222");
21         phoneBook.put("Михаил Н.", "(926) 222-333-111");
22         System.out.println("Исходная коллекция");
23         System.out.println(phoneBook);
24         display(phoneBook);
25     }
26 }
```

Результат работы программы:

```
Исходная коллекция
{Оксана Л.=(926) 111-222-333, Петр Ж.=(915) 333-111-222, Сергей М.=(929) 333-222-111, Михаил Н.=(926) 222-333-111}
Имя (key) = Оксана Л., Телефон (value) = (926) 111-222-333
Имя (key) = Петр Ж., Телефон (value) = (915) 333-111-222
Имя (key) = Сергей М., Телефон (value) = (929) 333-222-111
Имя (key) = Михаил Н., Телефон (value) = (926) 222-333-111
```

Для добавления записей используется метод `put`, которому в качестве параметров передается «ключ» (первый параметр - имя) и «значение» (второй параметр - телефон). Для хождения по хеш-таблице с помощью цикла `for-each`() используется метод `entrySet()`, которая возвращает объекта в виде коллекции `Set`.

Далее для каждого элемента `item` коллекции `Set` вызывается метод `getKey()` и `getValue()` вложенного интерфейса `Map.Entry`.

Продемонстрируем использование метода `get()` без использования методов интерфейса `Enumerator`.

Пример.

```
1  import java.util.*;
2  public class HashTableDemo
3  {
4      //экземплярный generic-метод
5      void display(Map ht){
6          //создаем вспомогательную коллекцию Set
7          //(множество) ключей из хэш-таблицы.
```

```
8        Set names = ht.keySet();
9        //цикл по множеству ключей
10       for(var element : names) {
11           System.out.println(element + ": " +
12                               ht.get(element));
13       }
14   }
15
16   public static void main(String args[]) {
17       Map balance = new Hashtable();
18       balance.put("Маша", new Double(3434.34));
19       balance.put("Михаил", new Double(123.22));
20       balance.put("Олег", new Double(1378.00));
21       HashTableDemo demo = new HashTableDemo();
22       demo.display(balance);
23       System.out.println();
24       //вносим 1 000 на аккаунт Маши
25       double account =
26         ((Double)balance.get("Маша")).doubleValue();
27       balance.put("Маша", new Double(account + 1000));
28       System.out.println("Новый баланс Маши: " +
29                           balance.get("Маша"));
30   }
31 }
```

Результат работы программы:

```
Маша: 3434.34
Олег: 1378.0
Михаил: 123.22

Новый баланс Маши: 4434.34
```

Извлечение записей выполняется методом `get()`, которому в качестве параметра передается ключ.

Выводы по применению методов коллекции Hashtable

Коллекция допускает дублирование ключей при этом если вносятся несколько значений с одинаковым ключом, то в объекте коллекции храниться только последняя по порядку добавления пара «ключ-значение».

Использование Hashtable без промежуточной конвертации данных в коллекцию типа Set возможно только если используются реализации методов устаревшего интерфейса Enumerable.

При использовании реализаций методов Map или методов вложенного интерфейса Map.Entry. Промежуточное преобразование данных к объекту коллекции Set обязательно (см. заголовки методов).

Однако, при использовании интерфейса Map.Entry невозможно разрабатывать generic-методы, т.к. требуется указывать *конкретные типы* ключа и значения, что может сильно снижать гибкость предлагаемых решений.

Коллекция HashMap

Эта коллекция является альтернативой HashTable. Основными *отличиями* HashMap является то, что:

- она не синхронизирована;
- у нее выше скорость выполнения операций;
- она позволяет использовать null как в качестве *ключа*, так и *значения*;
- для null-ключа hashCode() всегда равен нулю;
- порядок итерирования не совпадает с порядком добавления записей, но какой признак лежит в основе итерирования ***не*** установлено; вероятно, порядок расположения в памяти (и порядок итерирования) связан с размером памяти под хранение пары «ключ-значение».

Так же, как и HashTable:

- данная коллекция не является упорядоченной: порядок хранения элементов зависит от хэш-функции;
- не допускает дублей ключей (в коллекции остается последний по порядку добавления элемент с дублирующимся ключом).

Добавление элемента выполняется за константное время O(1), но время удаления, получения зависит от распределения хэш-функции. В идеале является константным, но может быть и линейным O(n).

Синтаксис *конструкторов* HashMap (количество параметров, их описание и примеры применения) *повторяют* конструкторы HashTable с точностью до замены названия.

Замечания.

Использует все те же интерфейсы, что `Hashtable` *с теми же особенностями по обязательной типизации пары «ключ-значение» в случае использования вложенного интерфейса* `Map.Entry`.

Все примеры, рассмотренные для `Hashtable` *работают и для* `HashMap` *с точностью до замены имени коллекции.*

Рассмотрим пример получения вписка всех ключей и значений.

Пример.

```
1   import java.util.*;
2
3   public class Main
4   {
5       //статический generic-метод
6       static void display(HashMap hm){
7         Set keys = hm.keySet();
8         System.out.println("Ключи: " + keys);
9         ArrayList values =
10              new ArrayList(hm.values());
11        System.out.println("Значения: " + values);
12      }
13
14      public static void main(String[] args) {
15          HashMap passportsAndNames = new HashMap();
16          passportsAndNames.put(212133,
17                          "Лидия Аркадьевна Бубликова");
18          passportsAndNames.put(162348,
19                          "Иван Михайлович Серебряков");
20          passportsAndNames.put(8082771,
21                          "Дональд Джон Трамп");
22           display(passportsAndNames);
23      }
24  }
```

Результат работы программы:

```
Ключи: [212133, 8082771, 162348]
Значения: [Лидия Аркадьевна Бубликова, Дональд Джон Трамп, Иван Михайлович Серебряков]
```

Объединение двух мап в одну осуществляется методом `putAll()` интерфейса `Map`. Его необходимо вызвать у первого объекта коллекции

HashMap, передавая второй в качестве аргумента, и элементы из второй будут добавлены в первую.

Пример.

```
import java.util.*;

public class Main
{
    public static void main(String[] args) {
        HashMap passport = new HashMap();
        passport.put(212133,
                         "Лидия Аркадьевна Бубликова");
        passport.put(8082771, "Дональд Джон Трамп");

        HashMap duty = new HashMap();
        //другие типы ключа и значения
        duty.put("Максим Олегович Архаров", 1005.79);
        duty.put("Алексей Андреевич Ермаков", 179.32);

        duty.putAll(passport);
        //после объединения выполняется упорядочение по
        //какому-то внутреннему алгоритму (признак
        //упорядочения не установлен), но это
        //гарантирует, что если поменять местами
        //passport и duty результат объединения будет
        //один и тот же
        System.out.println(duty);
    }
}
```

Результат работы программы:

```
{Алексей Андреевич Ермаков=179.32, Максим Олегович Архаров=1005.79, 212133=Лидия Аркадьевна Бубликова, 8082771=Дональд Джон Трамп}
```

HashMap можно синхронизировать с помощью вызова такого метода synchonizedMap() интерфейса коллекции:

```
Map ИмяСсылкиНаРезультат =
          Collections.synchonizedMap(ИмяИсхОбъекта);
```

Пример.

```
1  import java.util.*;
2
3  public class Main
4  {
5      public static void main(String[] args) {
6          HashMap passportMap = new HashMap();
7          passportMap.put(212133,
8                          "Лидия Аркадьевна Бубликова");
9          passportMap.put(917352,
10                         "Алексей Андреевич Ермаков");
11         Map synchrMap =
12                 Collections.synchronizedMap(passportMap);
13         System.out.println(synchrMap);
14     }
15 }
```

Результат выполнения программы:

```
{917352=Алексей Андреевич Ермаков, 212133=Лидия Аркадьевна Бубликова}
```

Коллекция LinkedHashMap

Коллекция `LinkedHashMap` — это очередная реализация хэш-таблицы. Здесь порядок итерирования равен порядку добавления элементов.

Элементы объекта данной коллекции имеют двунаправленные ссылки на следующий и предыдущий (аналогично `LinkedList`). В связи с этим увеличивается размер памяти, которую занимает объект коллекции по сравнению с объектом `HashMap`.

Особенности `LinkedList`:

- как и `HashMap`, `LinkedHashMap` выполняет базовые операции добавления, удаления и сохранения значений в объекте коллекции за постоянное время;
- допускает `null` в качестве ключа, а также значения;
- производительность `LinkedHashMap` с постоянным временем, вероятно, будет немного хуже, чем производительность `HashMap` с постоянным временем из-за дополнительной на поддержки двусвязного списка;

- итерация по элементам коллекции `LinkedHashMap` также занимает линейное время `O(n)`, аналогичное времени `HashMap`;
- однако производительность линейного времени `LinkedHashMap` во время итерации лучше, чем линейное время `HashMap`;
- реализация `LinkedHashMap` не синхронизирована.

Коллекция имеет четыре конструктора, совпадающие по синтаксису, набору и типу параметров с коллекциями `HashTable` и `HashMap`, но, кроме этого, имеет еще один конструктор:

```
LinkedHashMap<ТипКлюча, ТипЗначения>(int initialCapacity,
                                     float loadFactor,
                                     boolean accessOrder)
```

Параметры `initialCapacity` и `loadFactor` описаны ранее для конструкторов `HashTable`. Обратите внимание, что снижение эффективности работы с объектом коллекции из-за выбора чрезмерно высокого значения для начальной емкости (`initialCapacity`) менее серьезно для `LinkedHashMap`, чем для `HashMap`, поскольку время итерации для этого класса не зависит от емкости объекта.

Параметр `accessOrder` имеет специфическое значение. Передав `true`, включаем упорядочивание элементов в коллекции (порядок итерирования) от наименее недавно использованных до наиболее недавно использованных *после создания* (имеется в виду обращение методами `get()` и `put()`).

Если параметр `accessOrder` имеет значение `false`, то порядок доступа совпадает с порядком заполнения.

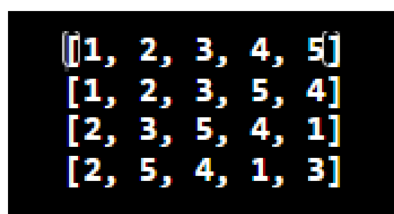
Пример.

```java
import java.util.*;

public class Main
{
    public static void main(String[] args) {
        LinkedHashMap map =
                new LinkedHashMap(16, .75f, true);
        map.put(1, null);
        map.put(2, null);
        map.put(3, null);
        map.put(4, null);
        map.put(5, null);

        Set<Integer> keys = map.keySet();
        //смотрим изменение порядка ключей после
        //обращений
        System.out.println(keys);
        map.get(4);
        System.out.println(keys);
        map.get(1);
        System.out.println(keys);
        map.put(3, 20);
        System.out.println(keys);
    }
}
```

Результат работы программы:



Обратите внимание, как меняется порядок элементов в наборе ключей, когда выполняем операции доступа к `LinkedHashMap`.

Проще говоря, любая операция доступа к карте приводит к такому порядку, что элемент, к которому осуществляется доступ, появляется последним, если перебор значений (итерирование) выполняется сразу после операции.

`LinkedHashMap` реализует методы интерфейса `Map`. Например, рассмотрим особенности реализации метода `putAll()` для объектов

LinkedHashMap, созданных с помощью конструкторов, в которых включена опция accessOrder.

Пример.

```
import java.util.*;

public class Main
{
    public static void main(String[] args) {
        LinkedHashMap map = new LinkedHashMap(16, .75f, true);
        map.put(1, null);
        map.put(2, null);
        map.put(3, null);
        map.put(4, null);
        map.put(5, null);

        Set<Integer> keys1 = map.keySet();
        //смотрим изменение порядка ключей после обращений
        map.get(4);
        map.get(1);
        System.out.println(keys1);

        LinkedHashMap duty =
                    new LinkedHashMap(16, .75f, true);
        //другие типы ключа и значения
        duty.put("Архаров", 1005.79);
        duty.put("Ермаков", 179.32);
        duty.get("Архаров");
        Set<Integer> keys2 = duty.keySet();
        //смотрим изменение порядка ключей после обращений
        System.out.println(keys2);
        map.putAll(duty);
        //после объединения переопределение порядка не
        //выполняется; результат операции в данном
        //случае зависит от порядка объединения
        //объектов и порядка элементов в каждом из
        //объектов
        System.out.println(map);
    }
}
```

Результат работы программы:

```
[2, 3, 5, 4, 1]
[Ермаков, Архаров]
{2=null, 3=null, 5=null, 4=null, 1=null, Ермаков=179.32, Архаров=1005.79}
```

После приведенного выше примера должно быть очевидно, что операция `putAll()` не упорядочивает результирующую коллекцию (даже если при создании обеих коллекций была включена опция `accessOrder`), а просто объединяет в том порядке в котором эти коллекции существовали на момент вызова метода `putAll()`.

Замечание.
При включенной опции `accessOrder(true)`:

- *обращение к объекту коллекции в целом (например, ее вывод на экран) не влияет на порядок обхода элементов.*
- *если создана копия коллекции и осуществлялись обращения к элементу копии, то это также не будет влиять на порядок итерирования в исходной коллекции. Только явные операции доступа к элементам самой коллекции изменят порядок.*

Кроме того коллекция `LinkedHashMap` имеет ряд собственных методов, среди которых можно выделить защищенный метод `removeEldestEntry(Map.Entry<K,V> eldest)`. Этот метод позволяет удалять самые старые записи из коллекции `LinkedHashMap` и контролировать таким образом количество записей в коллекции.

Поскольку метод защищен (имеет спецификатор доступа protected), то перед его использованием необходимо в классе наследнике `LinkedHashMap` его переопределить со спецификатором `public` либо это сделать в рамках анонимного класса.

Сперва продемонстрируем код переопределения метода в рамках класса наследника (класса-обертки) `LinkedHashMap`.

Пример.

```
1   import java.util.*;
2   
3   //класс-обертка для LinkedHashMap
4   class MyLinkedHashMap<K, V> extends LinkedHashMap<K, V> {
5       private static final int MAX_ENTRIES = 5;
6       //методы
7       public MyLinkedHashMap(int initialCapacity,
8                              float loadFactor,
9                              boolean accessOrder) {
10          super(initialCapacity,
11                loadFactor,
12                accessOrder);
13      }
```

```
14      @Override
15      protected boolean removeEldestEntry(Map.Entry eldest){
16          return this.size() > MAX_ENTRIES;
17      }
18  }
19
20  public class Main
21  {
22      public static void main(String[] args) {
23       LinkedHashMap map = new MyLinkedHashMap(16, .75f, true);
24          map.put(1, null);
25          map.put(2, null);
26          map.put(3, null);
27          map.put(4, null);
28          map.put(5, null);
29          Set keys = map.keySet();
30          System.out.println(keys);
31          map.put(6, null);
32          System.out.println(keys);
33          map.put(7, null);
34          System.out.println(keys);
35          map.put(8, null);
36          System.out.println(keys);
37      }
38  }
```

Результат работы программы:

```
[1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
[3, 4, 5, 6, 7]
[4, 5, 6, 7, 8]
```

Приведем пример синтаксиса с анонимным классом.

<u>Пример.</u>

```
1  import java.util.*;
2  public class LinkedHashMapDemo
3  {
4      //константа, ограничивающая число элементов колл.
5      private static final int MAX = 4;
6
7      public static void main(String[] args) {
8          LinkedHashMap lhm = new LinkedHashMap() {
9            //тело анонимного класса
```

```
10          @Override
11          protected boolean removeEldestEntry(Map.Entry e)
12          {
13             return this.size() > MAX;
14          }
15        };//конец анонимного класса
16          lhm.put(0, "Welcome");
17          lhm.put(1, "To");
18          lhm.put(2, "The");
19          lhm.put(3, "World");
20          System.out.println(lhm);
21          //дополнительный элемент с другими типами
22          lhm.put("Андрей", 79.3);
23          System.out.println(lhm);
24          //еще другой элемент
25          lhm.put(true, "1 миллион");
26          System.out.println(lhm);
27      }
28  }
```

Результат работы программы:

```
{0=Welcome, 1=To, 2=The, 3=World}
{1=To, 2=The, 3=World, Андрей=79.3}
{2=The, 3=World, Андрей=79.3, true=1 миллион}
```

По синтаксису применения метода `removeEldestEntry()` прослеживается некоторая аналогия с правилами использованием компаратора, хотя и не полная, т.к. компаратор необходимо явно передавать конструктору коллекции, а метод `removeEldestEntry()` используется неявно как метод `compareTo()` интерфейса `Compapable`.

Замечание.

Хотя в заголовке метода `removeEldestEntry()` стоит ссылка на интерфейс `Map.Entry`, но в данном случае переопределенный метод является generic-методом.

Синхронизацию объекта коллекции можно выполнить с помощью метода `synchronizedMap()` и следующей синтаксической конструкции:

```
Map m = Collections.synchronizedMap(new LinkedHashMap());
```

Интерфейс SortedMap

Интерфейс `SortedMap<K, V>` расширяет `Map<K, V>` и создает коллекцию, в которой все элементы отсортированы в порядке возрастания их ключей. `SortedMap` добавляет ряд методов:

- `K` **`firstKey`**`()` - возвращает ключ первого элемента объекта коллекции;
- `K` **`lastKey`**`()` - возвращает ключ последнего элемента объекта коллекции;
- `SortedMap<K, V>` **`headMap`**`(K end)` - возвращает ссылку типа `SortedMap` на объект коллекции, который содержит все элементы оригинального объекта вплоть до элемента с ключом `end`;
- `SortedMap<K, V>` **`tailMap`**`(K start)` - возвращает ссылку типа `SortedMap` на объект коллекции, который содержит все элементы оригинального объекта, начиная с элемента с ключом `start`;
- `SortedMap<K, V>` **`subMap`**`(K start, K end)` - ссылку типа `SortedMap` на объект коллекции, который содержит все элементы оригинального объекта коллекции от элемента с ключом `start` до элемента с ключом `end`.

Интерфейс NavigableMap

Интерфейс `NavigableMap<K, V>` расширяет интерфейс `SortedMap<K, V>` и обеспечивает возможность получения элементов объекта коллекции относительно других элементов этого же объекта.

Некоторые методы интерфейса:

- `Map.Entry<K, V>` **`firstEntry`**`()` - возвращает первый элемент объекта коллекции;
- `Map.Entry<K, V>` **`lastEntry`**`()` - возвращает последний элемент объекта коллекции;
- `NavigableSet<K>` **`navigableKeySet`**`()` - возвращает ссылку на объект `NavigableSet`, который содержит все ключи объекта коллекции;
- `NavigableMap<K, V>` **`headMap`**`(K up, boolean incl)` - возвращает ссылку `NavigableMap`, которая указывает на новый объект коллекции содержащую все элементы оригинального объекта коллекции вплоть от элемента с ключом `up`. Параметр `incl`

при значении true указывает, что элемент с ключом upp также включается в выходной набор;

- NavigableMap<K, V> **tailMap**(K low, boolean incl) - возвращает ссылку NavigableMap, которая указывает на новый объект коллекции содержащий все элементы оригинального объекта коллекции, начиная с элемента с ключом lowerBound. Параметр incl при значении true указывает, что элемент с ключом lowerBound также включается в выходной набор.

Коллекция TreeMap

Коллекция TreeMap<K, V> представляет соответствие в виде дерева. Он наследуется от класса AbstractMap и реализует интерфейс NavigableMap, а следовательно, также и интерфейсы SortedMap и Map.

В отличие от коллекций Hashtable, HashMap и LinkedHashMap в TreeMap все объекты автоматически сортируются по возрастанию их ключей с использованием принципа «natural ordering». Но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, который указывается в качестве параметра одного из конструкторов при создании объекта TreeMap.

Особенности TreeMap:

- коллекция не допускает null-ключи (например, Map), поэтому генерируется исключение NullPointerException;
- допускает null-значения с разными ключами;
- не синхронизирована.

Класс TreeMap имеет следующие конструкторы:

- TreeMap() - создает пустой объект в виде дерева:

Пример.

TreeMap~~<**ТипКлюча**, **ТипЗначения**>~~ tm = new TreeMap();

- TreeMap(Map<? Extends K, ? extends V> map) - создает дерево, в которое добавляет все элементы из соответствия уже созданного объекта с интерфейсом Map.

Пример

TreeMap~~<**ТипКлюча**, **ТипЗначения**>~~ tm = new TreeMap(hm);

- `TreeMap(SortedMap<K, ? extends V> smap)` - создает дерево, в которое добавляет все элементы из уже существующего объекта `smap` реализующего интерфейс `SortedMap` (собственно это другой уже созданный объект `TreeMap`).

Пример

TreeMap~~<**ТипКлюча**, **ТипЗначения**>~~ tm = new TreeMap();
TreeMap~~<**ТипКлюча**, **ТипЗначения**>~~ tmNext =
new TreeMap(tm);

- `TreeMap(Comparator<? Super K> comparator)` - создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированы пользовательским `comparator`-ом.

Как и ранее параметры ~~<**ТипКлюча**, **ТипЗначения**>~~ следует обязательно указывать если алгоритмически используются реализации методов интерфейса `Map.Entry`.

Замечание.

*В любом случае объект класса TreeMap должен состоять только из **однотипных** записей «ключ-значение», т.к. происходит их упорядочивание при вставке в объект коллекции, а для этого используется компаратор, сравнивающий **только два одинаковых по типу** значения.*

Поэтому если необходимо сохранить несколько наборов «ключ-значение» с разными типами, то для каждого набора нужна свой отдельный объект коллекции.

Пример использования TreeMap с конструктором без параметров

Приведем пример использования конструктора без параметров в двух вариантах:

- с явным указанием типов *<ТипКлюча, ТипЗначения>*;
- без указания типов.

Пример.

```
1  import java.util.*;
2
3  //класс типа запись (record)
4  record Person(String name) {}
5
```

```
6   public class Program
7   {
8       //пример экземплярного generic-метода
9       void display(Map tm) {
10          Set keys = tm.keySet();
11          //цикл по множеству ключей
12          for(var element : keys) {
13              System.out.println(element + ": " +
14                                 tm.get(element));
15          }
16      }
17      //пример статического не generic-метода из-за
18      //Map.Entry
19      static void staticDisplay(Map<Integer, String> tm) {
20       for(Map.Entry<Integer,String> item : tm.entrySet()){
21          System.out.printf("Key: %d  Value: %s \n",
22                            item.getKey(), item.getValue());
23          }
24      }
25
26      public static void main(String[] args) {
27          TreeMap<Integer, String> states = new TreeMap();
28          states.put(10, "Germany");
29          states.put(2, "Spain");
30          states.put(14, "France");
31          states.put(3, "Italy");
32          System.out.println("Объект по ключу 2 :" +
33                             states.get(2));
34          staticDisplay(states);
35          Set keys = states.keySet();
36          System.out.println("Все ключи states методом " +
37                             "keySet(): " + keys);
38          System.out.println("Все ключи states методом " +
39                            "navigableKeySet(): " +
40                             states.navigableKeySet());
41          Collection values = states.values();
42          System.out.println("Все значения states: "+values);
43          System.out.println("Первый элемент states: " +
44                                       states.firstEntry());
45          System.out.println("Последний элемент states: " +
46                                       states.lastEntry());
47          System.out.println("\t- ключ последнего элемента:"+
48                             states.lastEntry().getKey()+" "+
49                             "\n\t- значение посл. элемен.:"+
50                             states.lastEntry().getValue());
51          System.out.println("Все элементы states после "+
```

```
52                          "элемента с ключом 4: " +
53                           states.tailMap(4));
54          System.out.println("Все элементы states до "+
55                          "элемента с ключом 10: " +
56                          states.headMap(10));
57          System.out.println("Новая коллекция people:");
58          Map people = new TreeMap();
59          people.put("1240i54", new Person("Tom"));
60          people.put("1564i55", new Person("Bill"));
61          people.put("4540i56", new Person("Nick"));
62          Program obj = new Program();
63          obj.display(people);
64      }
65  }
```

Результат работы программы:

```
Объект по ключу 2 :Spain
Key: 2  Value: Spain
Key: 3  Value: Italy
Key: 10  Value: Germany
Key: 14  Value: France
Все ключи states методом keySet(): [2, 3, 10, 14]
Все ключи states методом navigableKeySet(): [2, 3, 10, 14]
Все значения states: [Spain, Italy, Germany, France]
Первый элемент states: 2=Spain
Последний элемент states: 14=France
    - ключ последнего элемента:14
    - значение посл. элемен.:France
Все элементы states после элемента с ключом 4: {10=Germany, 14=France}
Все элементы states до элемента с ключом 10: {2=Spain, 3=Italy}
Новая коллекция people:
1240i54: Person[name=Tom]
1564i55: Person[name=Bill]
4540i56: Person[name=Nick]
```

Конструктор с компаратором в качестве параметра

Перейдем к рассмотрению примера использования компаратора в качестве параметра конструктора.

Пример.

```
1  import java.util.*;
2
3  //класс типа запись (record)
4  record Student(int number, String name, String address){}
```

```
 5
 6  //класс имплементирующий интерфейс Comparator
 7  class ExampleCompar implements Comparator<Student> {
 8      public int compare(Student a, Student b)
 9      {
10          return a.number() - b.number();
11      }
12  }
13
14  public class Program
15  {
16     //не generic-метод, хотя типы и не указаны явно
17     static TreeMap initStud() {
18        //вызов конструктора с компаратором
19        TreeMap treeMap = new TreeMap(new ExampleCompar());
20        treeMap.put(new Student(111, "Иван", "Лондон"), 2);
21        treeMap.put(new Student(131, "Антон", "Париж"), 3);
22        treeMap.put(new Student(121, "Серж", "Брест"), 1);
23        return treeMap;
24     }
25     //не generic-метод
26     static TreeMap initInteger() {
27        //вызов конструктора без параметров
28        TreeMap treeMap = new TreeMap();
29        treeMap.put(1, null);
30        treeMap.put(2, null);
31        treeMap.put(3, null);
32        return treeMap;
33     }
34     //экземплярный generic-метод – выводит на экран два
35     //разных объекта (с разными типами «ключ-значение»)
36     void display(TreeMap tm) {
37        Set keys = tm.keySet();
38        for(var element : keys) {
39           System.out.println(element.toString() + " " +
40                                      tm.get(element));
41         }
42     }
43
44      public static void main(String[] args) {
45          Program obj = new Program();
46          System.out.println("TreeMap с ключами Student");
47          obj.display(initStud());
48          System.out.println("\nTreeMap с ключами целыми "+
49                                "числами");
```

```
50          obj.display(initInteger());
51      }
52  }
```

Результат работы программы:

```
TreeMap с ключами Student
Student[number=111, name=Иван, address=Лондон] 2
Student[number=121, name=Серж, address=Брест] 1
Student[number=131, name=Антон, address=Париж] 3

TreeMap с ключами целыми числами
1 null
2 null
3 null
```

Использование конструктора, принимающего в качестве значения объект коллекции, реализующей интерфейс Map

Этот конструктор используется для инициализации объекта типа `TreeMap` элементами из уже созданного объекта коллекции, реализующей интерфейс `Map`. Элементы будут отсортированы с использованием естественного порядка ключей.

Пример.

```
1   import java.util.*;
2
3   public class Program
4   {
5       //статический метод (не generic)
6       static Map initHashMap() {
7           Map<Integer, String> hm = new HashMap();
8           hm.put(10, "Строка");
9           hm.put(20, "Объявление");
10          hm.put(30, "Сообщение");
11          hm.put(25, "Реклама");
12          hm.put(15, "Заголовок");
13          return hm;
14      }
15      //статический generic-метод
16      static void display(TreeMap tm) {
17        Set keys = tm.keySet();
```

```
18         for(var element : keys) {
19             System.out.println(element.toString() +" " +
20                                     tm.get(element));
21         }
22     }
23 
24     public static void main(String[] args)
25     {
26         //вызов конструктора со значение параметра в
27         //виде ссылки на интерфейс Map, которая
28         //передает объект коллекции HashMap
29         TreeMap tm = new TreeMap(initHashMap());
30         System.out.println("TreeMap: ");
31         display(tm);
32     }
33 }
```

Результат работы программы:

```
TreeMap:
10 Строка
15 Заголовок
20 Объявление
25 Реклама
30 Сообщение
```

Применение конструктора, инициализируемого объектом коллекции TreeMap

Этот конструктор используется для инициализации `TreeMap` с записями из уже отсортированного объекта этой же карты. Смысл в создании этого отдельного конструктора заключается в том, что при инициализации передается так же и компаратор исходного объекта.

Причем этот компаратор останется функционален и в новой коллекции.

Пример.

```
1 import java.util.*;
2 
3 //класс типа запись (record)
4 record Student(int number, String name, String address){}
```

```
5
6   //класс имплементирующий интерфейс Comparator
7   class ExampleCompar implements Comparator<Student> {
8      public int compare(Student a, Student b) {
9         return a.number() - b.number();
10     }
11  }
12
13  public class Program
14  {
15     //не generic-метод, хотя типы и не указаны явно
16     static TreeMap initStud() {
17        //вызов конструктора с компаратором
18        TreeMap treeMap = new TreeMap(new ExampleCompar());
19        treeMap.put(new Student(111, "Иван", "Лондон"), 2);
20        treeMap.put(new Student(131, "Антон", "Париж"), 3);
21        treeMap.put(new Student(121, "Серж", "Брест"), 1);
22        return treeMap;
23     }
24
25     //экземплярный generic-метод
26     void display(TreeMap tm) {
27        Set keys = tm.keySet();
28        for(var element : keys) {
29           System.out.println(element.toString() + " " +
30                              tm.get(element));
31         }
32     }
33
34      public static void main(String[] args) {
35          //вызов конструктора с параметром типа TreeMap;
36          //компаратор буде передан в новую коллекцию
37          //"по умолчанию"
38          TreeMap result = new TreeMap(initStud());
39          //добавление новго элемента в новую коллекцию
40          //для демонстрации, что компаратор
41          //инициализирующего обьекта остался рабочим
42          result.put(new Student(0,"Мохамед","Минск"),26);
43          Program obj = new Program();
44          obj.display(result);
45      }
46  }
```

Результат работы программы:

```
Student[number=0, name=Мохамед, address=Минск] 26
Student[number=111, name=Иван, address=Лондон] 2
Student[number=121, name=Серж, address=Брест] 1
Student[number=131, name=Антон, address=Париж] 3
```

Синхронизация коллекции

В случае необходимости коллекцию можно синхронизировать с помощью метода `synchronizedSortedMap()`. Лучше всего выполнять синхронизацию во время создания, чтобы предотвратить случайный несинхронизированный доступ к набору. Это можно выполнить с помощью следующей синтаксической конструкции:

```
SortedMap sm =
       Collections.synchronizedSortedMap(new TreeMap());
```

Интерфейс Set

Интерфейс `Set` расширяет интерфейс `Collection` и представляет неупорядоченный набор *уникальных* элементов. `Set` не добавляет новых методов, только вносит изменения в унаследованные методы.

Интерфейс `Set` накладывает дополнительные условия, помимо унаследованных от интерфейса `Collection`, на контракты всех конструкторов и на контракты методов `add()` , `equals()` и `hashCode()`. В частности, метод `add()` добавляет элемент в коллекцию и возвращает `true`, *только* если в коллекции еще нет такого элемента. Объявления для других унаследованных методов также включены в `Set`, но не содержат никаких дополнительных условий.

Является моделью математического понятия «множество».

Коллекция HashSet

Коллекция `HashSet` — реализация интерфейса `Set`. Класс `HashSet` представляет хеш-таблицу. Объект `HashSet` внутри использует объект `HashMap` для хранения данных. В качестве ключа используется

добавляемый элемент, а в качестве значения - объект-пустышка (new Object()).

Особенности HashSet:

- порядок элементов может ***не*** совпадать с порядком добавления;
- элемент null допускаются в HashSet;
- коллекция не синхронизирована.

Замечание.

HashSet хранит элементы в произвольном порядке, но зато быстро находит их. Подходит, если порядок разработчику не важен, но важна скорость поиска элемента.

Для создания объекта HashSet можно воспользоваться одним из следующих конструкторов:

- HashSet() – создает пустую хеш-таблицу.

  Пример.
  ```
  HashSet<Тип> ht = new HashSet();
  ```

- HashSet(int capacity) - параметр capacity указывает начальную емкость таблицы, которая по умолчанию равна 16.

  Пример.
  ```
  HashSet<Тип> ht = new HashSet(20);
  ```

- HashSet(int capacity, float loadFactor)- параметр loadFactor или коэффициент заполнения, как и ранее это значение которого должно быть в пределах от 0.0 до 1.0, указывает, насколько должна быть заполнена емкость объектами прежде чем произойдет ее расширение.

  Пример.
  ```
  HashSet<Тип> ht = new HashSet(20, 0.9);
  ```

- HashSet(Collection<? Extends E> col) - создает хеш-таблицу, в которую добавляет все элементы коллекции col.

  Пример.
  ```
  HashSet<Тип> ht = new HashSet();
  HashSet<Тип> htNew = new HashSet(ht);
  ```

<u>Замечание.</u>
Параметр ~~<Тип>~~ не обязателен.

<u>Пример.</u>

```
1  import java.util.*;
2
3  public class Program
4  {
5     //generic-метод, хотя типы и не указаны явно
6     static HashSet initHS() {
7       //вызов конструктора без параметра
8         HashSet hashSet = new HashSet();
9         hashSet.add("Картофель");
10        hashSet.add("Морковь"  );
11        hashSet.add(1);
12        hashSet.add(null);
13      return hashSet;
14    }
15
16    //экземплярный generic-метод
17    void display(HashSet hs) {
18      for(var element : hs) {
19        System.out.println(element);
20      }
21    }
22    //статический generic-метод, использующий итератор
23    static void iterDisplay(HashSet hs) {
24        Iterator iter = hs.iterator();
25        while(iter.hasNext()) {
26          System.out.println(iter.next());
27        }
28    }
29
30    public static void main(String[] args) {
31      HashSet result = initHS();
32      //добавляем элемент в объект коллекции
33      result. add(true);
34      //следующая запись не должна попасть в набор
35      result. add("Картофель");
36      //вывод размера множества
37      Program obj = new Program();
38      obj.display(result);
39      System.out.println("Размер hashSet = " +
40                         result.size());
```

```
41          //по желанию можно использовать и этот метод
42          //iterDisplay(result);
43      }
44  }
```

Результат работы программы:

```
null
1
Морковь
Картофель
true
Размер hashSet = 5
```

Если требуется выполнить синхронизацию объекта коллекции, то это лучше всего сделать во время создания, чтобы предотвратить случайный не синхронизированный доступ к множеству.

Формат синхронизации во время создания, например, с конструктором без параметров:

```
Set имяМнож = Collections.synchronizedSet(new HashSet());
```

Коллекция LinkedHashSet

Класс `LinkedHashSet` наследует `HashSet`, не добавляя никаких новых методов, и поддерживает двусвязный список элементов множества в том порядке, *в котором они вставлялись* (но работает медленнее `HashSet`). Это позволяет организовать упорядоченную итерационную вставку в набор.

Конструкторы `LinkedHashSet`:

- `LinkedHashSet()` - создание пустого набора с начальной емкостью `16` и со значением коэффициента загрузки `0.75` «по умолчанию»;
- `LinkedHashSet(Collection<? Extends E> c)` - созда ние множества из элементов коллекции `c`;
- `LinkedHashSet(int capacity)` - создание множества с указанной начальной емкостью и со значением коэффициента загрузки «по умолчанию» `0.75`;

- `LinkedHashSet(int capacity,float loadFactor)` - создание множества с указанными начальной емкостью и коэффициентом загрузки.

Замечание.

Пример работы с `LinkedHashSet` *можно взять из предыдущего раздела с заменой класса* `HashSet` *на класс* `LinkedHashSet`

Также, как и `HashSet`, `LinkedHashSet` не синхронизирована. Поэтому при использовании объектов данного класса в приложении с множеством потоков, часть из которых может вносить изменения в набор, следует на этапе создания выполнить синхронизацию аналогично `HashSet`, например, с конструктором без параметров:

```
Set имя=Collections.synchronizedSet(new LinkedHashSet());
```

Интерфейс SortedSet

Интерфейс `SortedSet` предназначен для создания коллекций, который хранят элементы в отсортированном виде (сортировка по возрастанию). Элементы упорядочиваются в соответствии с их естественным порядком или в порядке `Comparator`, обычно указываемом при создании отсортированного набора. `SortedSet` расширяет интерфейс `Set`, поэтому такая коллекция опять же хранит только уникальные значения. `SortedSet` предоставляет следующие абстрактные методы:

- `E` **`first`**`()` - возвращает первый элемент множества;
- `E` **`last`**`()` - возвращает последний элемент множества;
- `SortedSet<E>` **`headSet`**`(E end)` - возвращает объект `SortedSet`, который содержит все элементы исходного множества до элемента `end`;
- `SortedSet<E>` **`subSet`**`(E start, E end)` - возвращает объект `SortedSet`, который содержит все элементы исходного множества между элементами `start` и `end`;
- `SortedSet<E>` **`tailSet`**`(E start)` - возвращает объект `SortedSet`, который содержит все элементы исходного множества, начиная с элемента `start`.
- `Comparator<? Super E>` **`comparator`**`()` - возвращает компаратор, используемый для упорядочения элементов множества, или `null`, если этот набор использует естественный порядок своих элементов.

- default Spliterator<E> **spliterator**() - создает Spliterator в отсортированном наборе.

Интерфейс NavigableSet

Интерфейс NavigableSet расширяет интерфейс SortedSet и позволяет извлекать элементы на основании их значений. NavigableSet определяет множество методов, среди которых можно выделить следующие:

- NavigableSet<E> **descendingSet**() - возвращает ссылку NavigableSet на объект, который содержит все элементы первичного множества NavigableSet в обратном порядке;
- NavigableSet<E> **headSet**(E val, boolean incl) - возвращает ссылку NavigableSet на объект, который содержит все элементы первичного множества NavigableSet до val. Параметр incl при значении true, позволяет включить в выходной набор элемент val;
- NavigableSet<E> **tailSet**(E val, boolean incl) - возвращает ссылку NavigableSet на объект, который содержит все элементы первичного множества NavigableSet, после val. Параметр incl при значении true, позволяет включить в выходной набор элемент val;
- NavigableSet<E> **subSet**(E lowerBound,
  boolean lowerIncl,
  upperBound,
  boolean highIncl) - возвращает ссылку NavigableSet на объект, который содержит все элементы первичного множества NavigableSet от lowerBound до upperBound.

Коллекция TreeSet

Обобщенный класс TreeSet<E> представляет структуру данных в виде дерева, в котором все объекты хранятся в отсортированном виде по возрастанию. Предоставляет возможность управлять порядком элементов в коллекции при помощи объекта класса, имплементирующего Comparator, либо сохраняет элементы с использованием «natural ordering».

`TreeSet` является наследником класса `AbstractSet` и реализует интерфейс `NavigableSet`, а следовательно, и интерфейс `SortedSet` и `Set`.

Как и все коллекции, имплементирующие интерфейс `Set`, хранит только уникальные элементы. Коллекция не синхронизирована (не потокобезопасна).

Класс `TreeMap` имеет следующие конструкторы:

- `TreeSet()` - создает пустое дерево:

Пример.

```
TreeSet<Тип> tm = new TreeSet();
```

- `TreeSet(Collection<? extends E> col)` - создает дерево, в которое добавляет все элементы коллекции `col`.

Пример

```
Vector<Тип> v = new Vector();
TreeSet<Тип> tm = new TreeSet(v);
```

- `TreeSet(Comparator<? Super K> comparator)` - создает пустое множество, где все добавляемые элементы впоследствии будут отсортированы пользовательским `comparator`-ом.
- `TreeSet(SortedSet<E> set)` - создает дерево, в которое добавляет все элементы из уже существующего объекта `set` реализующего интерфейс `SortedSet` (собственно это другой уже созданный объект `TreeSet`).

Пример

```
TreeSet<Тип> tm = new TreeMap();
TreeSet<> tmNext = new TreeMap(tm);
```

Замечание.

В любом случае объект класса `TreeSet` должен состоять только из ***однотипных*** *значений, т.к. происходит их упорядочивание при вставке в объект коллекции, а для этого используется компаратор, сравнивающий* ***только два одинаковых по типу*** *значения.*

Поэтому если необходимо сохранить несколько значений разного типа, то для каждого множества нужно свое отдельное множество.

Пример использования **`TreeSet`** с конструктором без параметров

Приведем пример использования конструктора без параметров в двух вариантах:

- с явным указанием параметра *<Тип>*;
- без указания типа элемента множества.

Пример.

```
import java.util.*;

public class Program
{
    //пример экземплярного generic-метода
    void display(Set tm) {
        //цикл по множеству ключей
        for(var element : tm) {
            System.out.println(element);
        }
    }

    public static void main(String[] args) {
        TreeSet<String> states = new TreeSet();
        states.add("Italy");
        states.add("Spain");
        states.add("France");
        states.add("Germany");
        //добавление уже существующего элемента
        states.add("Italy");
        Program obj = new Program();
        obj.display(states);
        System.out.println("Первый объект states:" +
                           states.first());
        System.out.println("Последний объект states:" +
                           states.last());

        System.out.println("Все значения от F до I " +
                           "включительно " +
                    states.subSet("F", true, "I", true));
        System.out.println("Все значения после I " +
                          states.tailSet("I"));
        System.out.println("Все значения до I " +
                          states.headSet("I"));
        System.out.println("Исх. множество в обр. порядке"+
                          states.descendingSet());
        System.out.println("\nНовое множество чисел");
```

```
38          TreeSet numbers = new TreeSet();
39          numbers.add(new Integer(1));
40          numbers.add(new Integer(5));
41          numbers.add(new Integer(10));
42          obj.display(numbers);
43      }
44  }
```

Результат работы программы:

```
France
Germany
Italy
Spain
Первый объект states:France
Последний объект states:Spain
Все значения от F до I включительно [France, Germany]
Все значения после I [Italy, Spain]
Все значения до I [France, Germany]
Исх. множество в обр. порядке[Spain, Italy, Germany, France]

Новое множество чисел
1
5
10
```

Конструктор с компаратором в качестве параметра

Перейдем к рассмотрению примера использования компаратора в качестве параметра конструктора.

Пример.

```
1   import java.util.*;
2   
3   //класс типа запись (record)
4   record Student(int number, String name, String address){}
5   
6   //класс имплементирующий интерфейс Comparator
7   class ExampleCompar implements Comparator<Student> {
8       public int compare(Student a, Student b)
9       {
10          return a.number() - b.number();
11      }
12  }
```

```
13
14  public class Program
15  {
16      //не generic-метод, хотя типы и не указаны явно
17      static TreeSet initStud() {
18          //вызов конструктора с компаратором
19          TreeSet treeSet = new TreeSet(new ExampleCompar());
20          treeSet.add(new Student(111, "Иван", "Лондон"));
21          treeSet.add(new Student(131, "Антон", "Париж"));
22          treeSet.add(new Student(121, "Серж", "Брест"));
23          return treeSet;
24      }
25      //не generic-метод
26      static TreeSet initInteger() {
27          //вызов конструктора без параметров
28          TreeSet treeSet = new TreeSet();
29          treeSet.add(1);
30          treeSet.add(2);
31          treeSet.add(3);
32          return treeSet;
33      }
34      //экземплярный generic-метод использующий итератор
35      void display(TreeSet tm) {
36          Iterator iter = tm.iterator();
37          while(iter.hasNext()) {
38              System.out.println(iter.next());
39           }
40      }
41
42       public static void main(String[] args) {
43          Program obj = new Program();
44          System.out.println("TreeMap с ключами Student");
45          obj.display(initStud());
46          System.out.println("\nTreeMap с ключами целыми "+
47                                 "числами");
48          obj.display(initInteger());
49       }
50  }
```

Результат работы программы:

```
TreeMap с ключами Student
Student[number=111, name=Иван, address=Лондон]
Student[number=121, name=Серж, address=Брест]
Student[number=131, name=Антон, address=Париж]

TreeMap с ключами целыми числами
```

```
1
2
3
```

<u>Замечание.</u>

Как видно из примеров идеология `TreeSet` в целом повторяет идеологию `TreeMap`. Это выражается непосредственно во всех рассмотренных примерах.

Достаточно в примерах рассмотренных ранее для `TreeMap` заменить метод `put()` на `add()`, а также внести изменения касающиеся того что в `TreeSet` храниться только значение (а не пара «ключ-значение») и с точностью до методов интерфейсов `SortedSet` и `NavigableSet` все примеры для `TreeSet` будут повторять уже известные для `TreeMap`.

В случае необходимости коллекцию можно синхронизировать как и ранее для коллекций имплементирующих интерфейс `Set` с помощью метода `synchronizedSet()` с точность до замены конструктора:

```
Set имя = Collections.synchronizedSet(new TreeSet());
```

Литература

1. Блинов, И.Н. Java from EPAM / И.Н. Блинов, В.С. Романчик. - Минск: Четыре четверти, 2020. - 560 с.
2. Руководство по языку программирования Java/ METANIT.COM - [Электронный ресурс], URL: https://metanit.com/java/tutorial/ (дата обращения: 08.06.22)
3. Самоучитель по Java с нуля/ Vertex Academy - [Электронный ресурс], URL: https://vertex-academy.com/tutorials/ru/samouchitel-po-java-s-nulya/ (дата обращения: 08.06.22)
4. javascopes.com - [Электронный ресурс], URL: http://javascopes.com (дата обращения: 08.06.22)
5. JavaRush - [Электронный ресурс], URL: https://javarush.ru/ (дата обращения: 08.06.22)
6. Справочник по Java Collections Framework / Хабр - [Электронный ресурс], URL: https://habr.com/ru/post/237043/ (дата обращения: 08.06.22)
7. Collections in Java / GeeksforGeeks - [Электронный ресурс], URL: https://www.geeksforgeeks.org/collections-in-java-2/?ref=lbp (дата обращения: 08.06.22)
8. Java Collections / Baeldung - [Электронный ресурс], URL: https://www.baeldung.com/java-collections (дата обращения: 08.06.22)

Учебное издание

Кравчук Александр Степанович
Кравчук Анжелика Ивановна
Кремень Елена Васильевна

Язык Java. Коллекции, реализующие интерфейсы Queue, Deque, Map, Set

Учебные материалы для студентов специальности 1-31 03 08 «Математика и информационные технологии (по направлениям)»

В авторской редакции

Ответственный за выпуск *Е. В. Кремень*

Подписано в печать 22.12.2022. Формат 60×84/16. Бумага офсетная.
Усл. печ. л. 4,65. Уч.- изд. л. 4,24. Тираж 50 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя, распространителя печатных изданий № 1/270 от 03.04.2014.
Пр. Независимости 4, 220030, Минск.

Отпечатано с оригинал-макета заказчика
на копировально-множительной технике
механико-математического факультета
Белорусского государственного университета.
Пр. Независимости 4, 220030, Минск.