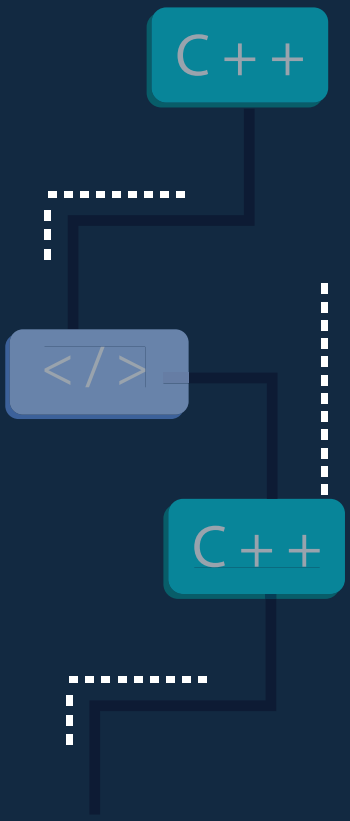

top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Язык программирования JAVA

Урок №9

JavaCollection Framework

Содержание

1. Введение в коллекции

Что такое коллекции, и зачем они нужны? Давайте обсудим этот вопрос, прежде чем начнем разбирать существующие коллекции и рассматривать примеры их использования. В каждом языке программирования есть такой способ организации данных, как массив. Массив зарекомендовал себя как простой и эффективный способ хранения множества значений. Все знают о чрезвычайно высокой скорости доступа к элементам массива. Зачем же нужны еще какие-то коллекции?

Дело в том, что наряду с очевидными достоинствами у массива есть и ряд недостатков. Прежде всего, это невозможность изменения его размера. Также следует отметить еще одну особенность массива, которая иногда является недостатком – это способ его размещения в памяти. Вы знаете, что все элементы массива хранятся в памяти одним непрерывным куском, непосредственно друг за другом. И если вы объявили массив ar[1000], то для его создания необходимо наличие в оперативной памяти непрерывной свободной области размером в 1000 ячеек. Именно такое размещение элементов массива в памяти обусловливает высокую скорость доступа к его элементам. Все элементы расположены рядом, и перемещения между ними легко выполнимы. В чем здесь недостаток?

Если вам понадобится очень большой массив, может произойти так, что в памяти не найдется непрерывной свободной области для его размещения. Таким образом, требование непрерывного размещения элементов массива в памяти, с одной стороны, является достоинством, так как обеспечивает быстрый доступ к ним. С другой сторо-

ны, оно же является и недостатком, так как ограничивает размер создаваемого массива.

Теперь вы понимаете, для чего могут быть нужны коллекции. В некоторых случаях недостатки массивов могут быть критичными, и тогда надо использовать коллекции, у которых таких недостатков нет. Другими словами, коллекции – это такие способы организации данных, которые в некоторых случаях оказываются более эффективными, чем массив.

Давайте сделаем еще один вывод из предыдущего абзаца. Оказывается, способ хранения элементов коллекции в оперативной памяти определяющим образом влияет на ее свойства.

2. Java Collection Framework

Для работы с коллекциями в Java, начиная с версии 1.2, создан Java Collection Framework. Его задача – стандартизировать работу с коллекциями. Фреймворк состоит из интерфейсов, классов и алгоритмов. В интерфейсах содержатся наборы методов для работы с коллекциями. Классы наследуют базовым интерфейсам и индивидуальным образом реализуют методы, полученные в наследство от них. Алгоритмы реализованы в виде статических методов и предоставляют общую для коллекций функциональность, такую как поиск, сортировка, перемешивание. Структура фреймворка JCF состоит из двух различных иерархий, происходящих от интерфейсов Collection и Map. В иерархии, производной от Collection, реализованы различные коллекции для работы с одиночными элементами. В иерархии, производной от Map, реализованы коллекции пар "ключ-значение". Рассмотрим структуру этих двух иерархий.

3. Интерфейс Collection

Строго говоря, интерфейс Collection не является самым базовым в своей иерархии. Он сам наследует интерфейсу Iterable, добавляющему функциональность итераторов. В интерфейсе Collection собраны такие понятные методы, как size(), isEmpty(), add(), remove(), clear() и другие. Интерфейсу Collection наследуют три других интерфейса: List, Set и Queue. Каждый из этих трех интерфейсов является базовым для различных наборов коллекций. Рассмотрим их подробнее.

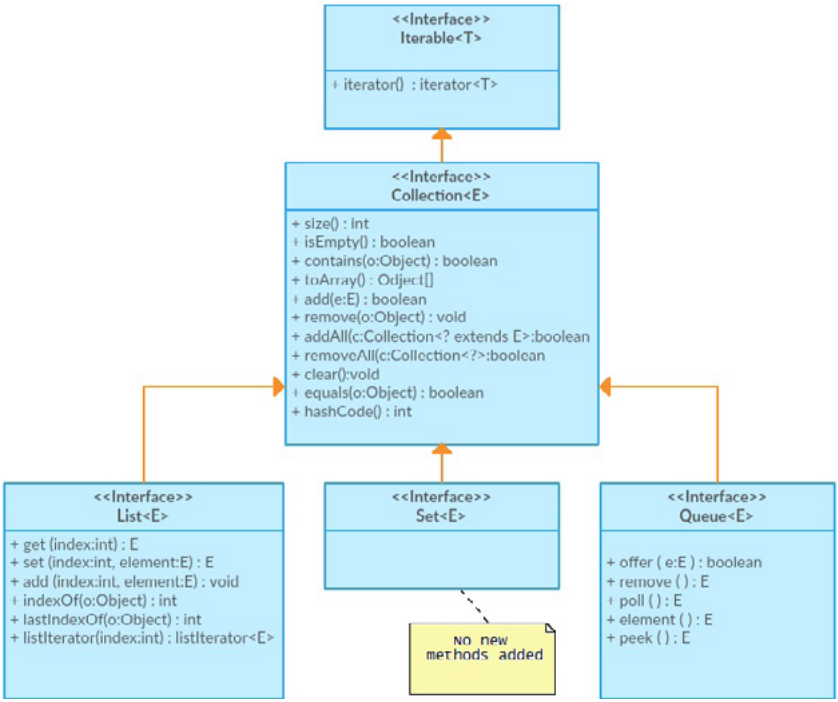

Рис. 1. Интерфейс Collection

List

List является базовым типом для коллекций, называемых последовательностями (*sequence*). Такие коллекции являются неупорядоченными и допускают наличие элементов с равными значениями. Элементы в последовательностях пронумерованы и допускают обращение по индексу.

На приведенной ниже UML-диаграмме видно, что интерфейсу List наследуют два класса: ArrayList и LinkedList. Для наглядности на этой диаграмме не указаны еще некоторые промежуточные интерфейсы и абстрактные классы.

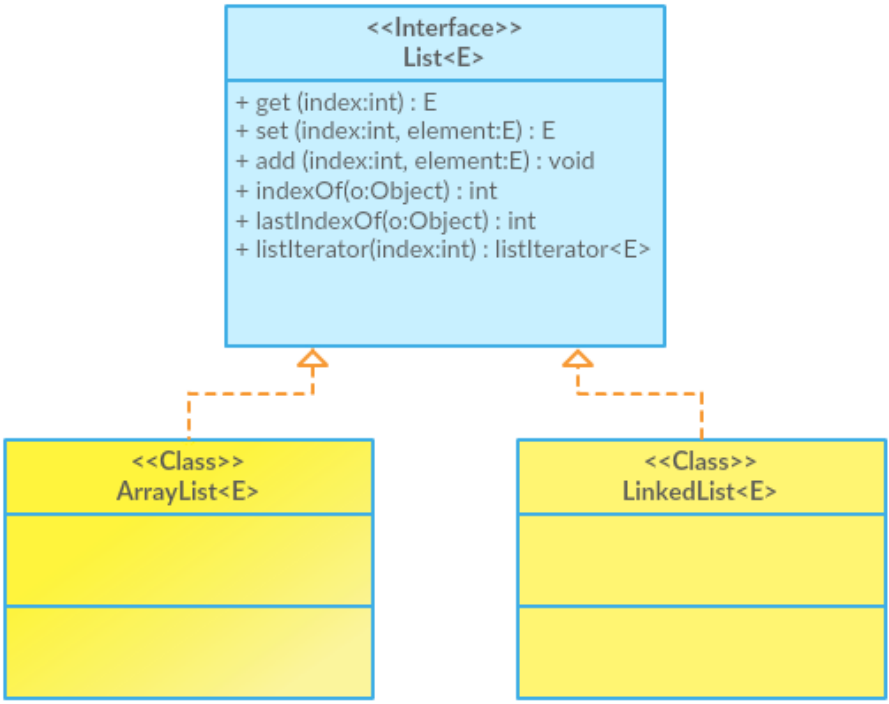

Рис. 2. Интерфейс List

Объекты классов ArrayList и LinkedList представляют собой коллекции. Давайте рассмотрим их характеристики и примеры использования.

ArrayList

Элементы в этой коллекции располагаются в памяти, как в массиве – непрерывно друг за другом. Поэтому доступ к элементам ArrayList очень быстрый. Однако, в отличие от массива, ArrayList может изменять свой размер. Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html.

Мы рассмотрим самые основные особенности этой коллекции. У ArrayList есть свойство capacity (емкость), которое задает количество ячеек памяти, выделенных для коллекции. Не надо путать это свойство с количеством элементов коллекции, которое можно узнать, вызвав метод size().

Если при добавлении в ArrayList очередного элемента окажется, что capacity уже исчерпана, будет выполнено автоматическое увеличение размера коллекции. Это увеличение выполняется таким образом. Сначала создается новая коллекция, размером (N * 3) / 2 + 1, где N – текущее значение capacity. В эту новую коллекцию переносятся данные из заполненной коллекции, и затем добавляется новый элемент. Старая коллекция удаляется.

Эти действия выполняются всякий раз при переполнении текущего значения capacity. Как видите, недостаток массива, запрещающий изменять его размер, преодолен, но ценой некоторых ресурсных и временных затрат.

Элементы в ArrayList можно удалять. При удалении какого-либо элемента, все элементы, располагающиеся справа от него, смещаются на одну позицию влево. Обратите внимание, что capacity коллекции при этом не изменяется. Сразу надо сделать оговорку относительно возможности явно увидеть значение емкости. В отличие от C#, где класс ArrayList содержит свойство capacity и позволяет увидеть его значение, в Java ситуация иная. Разработчики Java говорят, что нам не надо знать конкретное значение capacity, и не предоставляют нам возможность увидеть его. Хотя в Java документации по ArrayList можно увидеть, что изначально ArrayList создается со значением capacity, равным 10, это значение, вообще говоря, может отличаться в реализации разных JVM.

Тип элементов коллекции ArrayList может быть любым. Он задается при создании коллекции с помощью generic:

```
ArrayList<String> list1=new ArrayList<String>();
//Creating arraylist of String

ArrayList<Fish> list2=new ArrayList<Fish>();
//Creating arraylist of type Fish

ArrayList<Object> list3=new ArrayList<Object>();
//Creating arraylist of Object
```

ArrayList имеет три вида конструкторов:

```
ArrayList() //создает пустую коллекцию;
ArrayList(Collection <? extends E> c) //создает
//коллекцию, в которую при создании добавляются все
//элементы коллекции c;
```

```
ArrayList (int capacity) // создает коллекцию с
                         // с начальной емкостью
                         // capacity;
```

Обратите внимание на третий конструктор. Вы помните, что ArrayList позволяет изменять количество элементов в коллекции и делает это автоматически, при переполнении емкости. Однако такое увеличение довольно затратно по времени и ресурсам. Поэтому, если вы можете сделать предположение относительно емкости создаваемой вами коллекции, то лучше задать желаемую емкость при создании, чтобы уменьшить количество автоматических увеличений размера коллекции.

Рассмотрим примеры кода, демонстрирующие основные действия с ArrayList. Сначала создадим пустую коллекцию. Мы указываем в generic желаемый тип элементов – String. Обратите внимание, что начиная с Java 7 в правой части строки создания generic можно оставлять пустым.

```
ArrayList<String> al = new ArrayList<>(); //create
                       // empty collection of String
```

Используя метод add(), добавим в коллекцию названия пяти стран. К элементам ArrayList можно обращаться по индексу, учитывая, что начальный индекс равен 0. Таким образом, вызов метода al.get(1) вернет второй элемент коллекции – "Bulgaria".

```
al.add("Argentina");
al.add("Bulgaria");
al.add("Canada");
```

```
al.add(“Denmark”);
al.add("Narnia");
System.out.println("Collection:"+al);
System.out.println("Collection's size:"+al.size());
System.out.println(al.get(1));
```

ArrayList допускает наличие одинаковых элементов. Предположим, мы не хотим дублировать в нашей коллекции названия стран. Поэтому такой вызов метода al.contains("England") позволит нам узнать, есть ли в списке элемент "England". Если такого элемента нет, добавим его вместо несуществующей страны "Narnia". Для добавления используем метод al.set(4,"England"), который занесет значение "England" в элемент с индексом 4. Обратите внимание, что метод set() может изменять значения только существующих элементов. Если вы в первом параметре метода set() укажете индекс элемента, которого еще нет в коллекции, вы получите ошибку.

```
if(!al.contains("England"))
{
    System.out.println("England is not in Collection");
    al.set(4,"England");
}
// al.set(5,"France"); //causes error
System.out.println("Collection:"+al);
System.out.println("Collection's size:"+al.size());
System.out.println(al.indexOf("England"));
```

Если вам надо узнать индекс элемента с заданным значением, вы можете использовать метод al.indexOf ("England"), которому надо передать значение требуемо-

го элемента. Например, давайте переименуем "England" в "United Kingdom". Для этого можно использовать метод set(), но ему надо указать индекс изменяемого элемента. Здесь нам и пригодится метод indexOf().

```
int ie = al.indexOf("England");
al.set(ie, "United Kingdom");
System.out.println("Collection:"+al);
System.out.println("Collection's size:"+al.size());
```

Очень часто, для выполнения каких-либо действий, возникает необходимость перебирать элементы коллекции в цикле. В вашем распоряжении все циклы языка Java. Однако, отметьте такую особенность: перебор коллекции в каком-либо цикле не позволяет удалять элементы в перебираемой коллекции. Это плохая новость. А вот и хорошая: перебор коллекции с помощью итератора позволяет удалять элементы коллекции.

```
System.out.println("Collection Using For Loop:");
for (int i = 0; i < al.size(); i++) {
    System.out.println(al.get(i));
}

System.out.println("\nCollection Using While Loop:");
int i = 0;
while (i < al.size()) {
    System.out.println(al.get(i));
    i++;
}

System.out.println("\nCollection Using Advanced
                    For Loop:");
for (Object a : al) {
```

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

System.out.println("\nCollection Using Iterator:");
Iterator<String> iter = al.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

Если для вас является важным оптимальное использование памяти, можете сделать так, чтобы ваша коллекция не занимала в памяти бóльшую емкость, чем количество элементов в ней. Для этого надо использовать метод trimToSize(). Выполнение этого метода приведет к усечению емкости коллекции до текущего количества элементов. Понятно, что первое же добавление в коллекцию нового элемента после усечения, сразу же вызовет автоматическое увеличение ее размера.

```
al.trimToSize();
```

Независимо от того, каким конструктором вы создали свою коллекцию, вы можете установить ее емкость в требуемое вам значение, вызвав метод ensureCapacity(N), где N – требуемое значение емкости. Сделаем емкость нашей коллекции равной 100.

```
al.ensureCapacity(100);
```

Очень часто бывают ситуации, когда на этапе формирования коллекции полезно работать с ArrayList, а в дальнейшем, когда коллекция уже сформирована, хотелось бы вернуться к массиву. Хотя бы потому, что массив

все-таки быстрее, чем ArrayList. Такое преобразование можно выполнить с помощью метода toArray().

```
String[] ar = al.toArray();
```

LinkedList

Рассмотрим второй класс, реализующий интерфейс List – LinkedList. Этот класс представляет собой двухсвязный список. Кроме интерфейса List класс LinkedList еще реализует интерфейсы List, Dequeue и Queue. Поэтому эта коллекция предоставляет функциональность FIFO и LIFO. Список допускает наличие одинаковых элементов и элементов со значением null.

В таком списке каждый элемент содержит непосредственно данные и два указателя: один на следующий элемент, другой – на предыдущий. Элементы списка размещаются в памяти произвольно и, в отличие от массива, не занимают непрерывный сегмент памяти. Поэтому доступ к произвольному элементу списка гораздо медленнее, чем в массиве.

Список позволяет быстрее, чем массив выполнять добавление и удаление элементов в середине. Поэтому он лучше подходит в тех ситуациях, когда вам требуется интенсивно добавлять или удалять элементы коллекции. Однако для использования этого преимущества надо применять специальный подход. Об этом поговорим ниже.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html.

LinkedList имеет два вида конструкторов:

```
LinkedList ()                    создает пустую коллекцию;

LinkedList (Collection <? extends E> c) создает
                                 коллекцию, в которую при
                                 создании добавляются все
                                 элементы коллекции c.
```

Для добавления элементов в конец списка можно использовать методы add(value) и addLast(value), а для добавления в начало списка – addFirst(value). Все эти методы выполняются за постоянное время. Другими словами, время выполнения этих действий не зависит от размера коллекции, в то время как есть действия, время выполнения которых увеличивается с увеличением размера коллекции.

```
LinkedList<String> ll=new LinkedList<String>();
ll.add("one");
ll.add("two");
ll.add("three");
ll.add("four");
ll.add("five");
System.out.println("List:"+ll);
ll.addLast("six");
ll.add(3,"three");
ll.addFirst("zero");
System.out.println("List:"+ll);
```

Для добавления элементов в произвольное место списка можно использовать метод add(index, value). Здесь index указывает на элемент, перед которым будет вставлено value. Поиск этого элемента ведется последовательным

перебором либо от начала списка, либо от конца – в зависимости от того, расположен index ближе к началу или к концу списка. Такая вставка выполняется медленнее, чем вставка элементов в начало или конец списка, и тем медленнее, чем больше размер коллекции.

Для удаления элементов в конце и в начале списка используются методы removeLast() и removeFirst(), соответственно. Такое удаление выполняется за постоянное время. Удалять элементы в произвольном месте списка можно по индексу или по значению элемента методами remove(index) и remove(value). Удаление в произвольном месте списка выполняется медленнее, чем в начале и конце списка. При удалении по значению удаляется только первый найденный элемент.

```
ll.remove("three");
System.out.println("List:"+ll);
```

Перебирать список можно в обычном цикле.

```
System.out.println("Loop for:");
for (int i = 0; i < ll.size(); i++)
{
    System.out.println(ll.get(i));
}
```

Однако гораздо эффективнее использовать для перебора элементов итератор ListIterator. Этот итератор можно создавать таким образом, что сразу после создания он будет указывать на первый элемент списка или же на произвольный элемент списка по индексу.

```
ListIterator<String> it_beg = ll.listIterator();
System.out.println("Loop forward:");
while(it_beg.hasNext())
{
    System.out.println(it_beg.next());
}
System.out.println("Loop backward:");
while(it_beg.hasPrevious())
{
    System.out.println(it_beg.previous());
}
ListIterator<String> it_ind = ll.listIterator(4);
System.out.println("Loop from index:");
while(it_ind.hasNext())
{
    System.out.println(it_ind.next());
}
```

Если требуется перебирать коллекцию с конца списка, то можно создать итератор, указывающий на последний элемент списка, вызвав метод descendingIterator().

```
Iterator<String> it_desc = ll.descendingIterator();
System.out.println("Loop with descending Iterator:");
while(it_desc.hasNext())
{
    System.out.println(it_desc.next());
}
```

Если коллекция была изменена после создания итератора методами, не принадлежащими этому итератору, то итератор становится недействительным и обращение к нему приведет к ConcurrentModificationException. Если же изменять коллекцию методами самого итератора,

такого не произойдет. У итератора есть методы add(), remove() и set(), предназначенные для добавления, удаления и изменения элемента. Эти методы следует вызывать после вызова методов итератора next() или previous().

```
it_ind.set("6");
it_ind.add("7");
```

В начале урока мы говорили, что в состав фреймворка JCF входят алгоритмы. Рассмотрим пример использования алгоритма. Давайте отсортируем нашу коллекцию.

```
Collections.sort(ll);
System.out.println("Sorted list:");
for(String s:ll)
{
    System.out.println(s);
}
```

Вывод этого кода будет таким:

```
Sorted list:
6
7
five
four
one
three
two
zero
```

Видно, что строковые элементы коллекции ll упорядочены по возрастанию в лексикографическом порядке.

Статический метод sort(), вызываемый от имени класса Collections, является представителем алгоритмов. В этом классе есть еще целый ряд алгоритмов, предназначенных для выполнения типичных действий с коллекциями:

```
Collections.copy()      копирует одну коллекцию
                        в другую;
Collections.max()       возвращает максимальный
                        элемент коллекции;
Collections.min()       возвращает минимальный элемент
                        коллекции;
Collections.reverse()   обращает порядок элементов
                        коллекции с конца в начало;
```

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html.

Сравнение ArrayList и LinkedList

Отличия этих двух коллекций обусловлены способами хранения элементов. ArrayList хранит свои элементы как массив, а LinkedList – как список. Это приводит к тому, что доступ по индексу у первой коллекции намного быстрее, чем у второй. Однако операции добавления и удаления элементов в произвольном месте коллекции в LinkedList выполняются быстрее. Правда, для этого надо использовать ListIterator, который позволит быстро добраться до требуемого места коллекции, а не перебирать все элементы от хвоста или головы списка.

Если говорить о потребляемой памяти, то LinkedList требует больше памяти, чем ArrayList, по причине использования ссылок на соседние элементы.

Set

Set является базовым для коллекций, являющихся неупорядоченными и не допускающих наличия элементов с равными значениями. Элементами с равными значениями называются такие элементы коллекции el1 и el2, для которых выполняется условие el1.equals(el2). Такие коллекции называются множествами.

На приведенной ниже UML-диаграмме видно, что интерфейсу Set наследуют три класса: HashSet, LinkedHashSet и TreeSet. Для наглядности на этой диаграмме не указаны еще некоторые промежуточные интерфейсы и абстрактные классы.

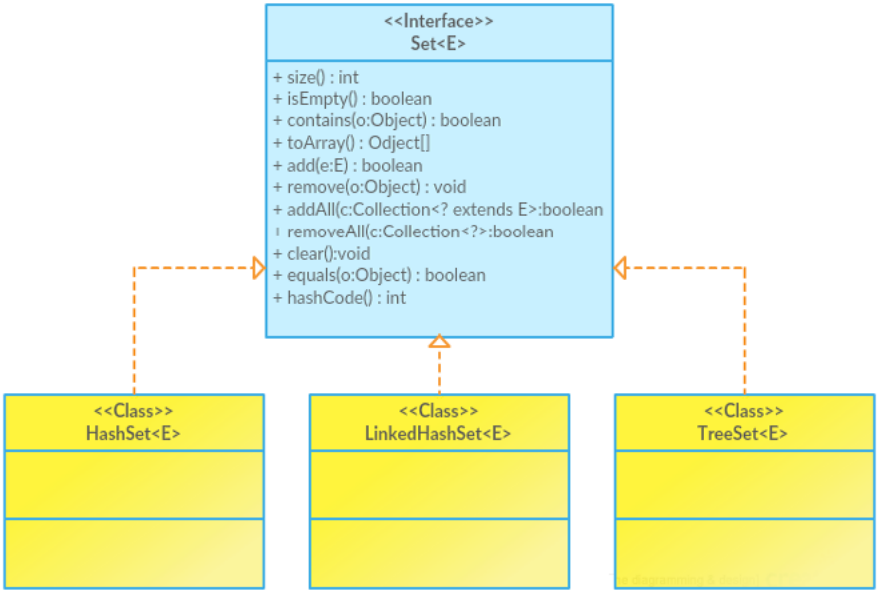

Рис. 3. Интерфейс Set

Рассмотрим эти три класса подробнее.

HashSet

Коллекция HashSet оптимизирована для быстрого поиска уникальных элементов. Для каждого добавляемого элемента эта коллекция вычисляет хеш код, который затем использует как ключ для доступа к этому элементу. Сам хеш код увидеть невозможно. Использование хеширования для создания ключей к элементам этой коллекции позволяет выполнять такие методы, как add(), contains(), remove() и size() за постоянное время даже для очень больших коллекций. Другими словами, время выполнения этих операций не зависит от величины коллекции. Среди трех реализаций интерфейса Set эта коллекция является самой быстрой. Если вы собираетесь хранить в ней объекты своего класса, то позаботьтесь о переопределении в этом классе методов hashCode() и equals(). Если этого не сделать, при добавлении двух одинаковых объектов такого класса в коллекцию будет вызываться метод hashCode() класса Object, который вернет разные хеш коды для одинаковых объектов. HashSet не обеспечивает упорядоченности своих элементов.

Методы этого класса эквивалентны методам класса ArrayList, с тем отличием, что метод add(value) добавляет элемент в коллекцию, если такого элемента в ней еще нет, и возвращает true. А в случае, если добавляемый элемент в коллекции уже присутствует, этот метод его не добавляет и возвращает false.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html.

HashSet имеет такие конструкторы:

```
HashSet ()     создает пустую коллекцию;
HashSet (Collection <? extends E> c)  создает
              коллекцию, в которую при создании
              добавляются все элементы коллекции c;
HashSet (int capacity)  создает коллекцию с начальной
              емкостью capacity;
HashSet (int capacity, float loadfactor)  создает
              коллекцию с начальной емкостью
              capacity и с заданным фактором загрузки.
```

Фактор загрузки – это значение, которое определяет, при каком уровне заполнения коллекции ее емкость автоматически увеличивается. Когда количество элементов в коллекции превышает произведение фактора загрузки и текущей емкости, выполняется автоматическое увеличение коллекции.

Рассмотрим примеры кода. Не будем останавливаться на обычных методах добавления, извлечения или изменения элементов коллекции, поскольку они такие же, как и у класса ArrayList. Давайте уделим внимание созданию HashSet из объектов пользовательского класса.

Пусть у нас есть такой класс:

```
public class Fish
{
    private String name;
    private double weight;
    private double price;
    public Fish(String name, double weight, double price)
    {
        this.name=name;
```

```
        this.weight=weight;
        this.price=price;
    }

    @Override
    public String toString()
    {
        return this.name+"  weight:"+this.weight+"
        price:"+this.price;
    }
}
```

Создадим коллекцию из его объектов:

```
HashSet<Fish> fishes = new HashSet<>();
fishes.add(new Fish("eel",1.5,120));
fishes.add(new Fish("salmon",2.5,180));
fishes.add(new Fish("carp",3.5,80));
fishes.add(new Fish("trout",2.2,150));

System.out.println("Collection:"+fishes);
System.out.println("Collection's size:"+fishes.size());
```

Вывод этого кода выглядит таким образом:

```
Collection:[carp  weight:3.5  price:80.0,
trout  weight:2.2  price:150.0, eel  weight:1.5
price:120.0, salmon  weight:2.5  price:180.0]
Collection's size:4
```

Все выглядит ожидаемо. Мы создали коллекцию из объектов нашего класса и увидели ее элементы. Обратите внимание, что элементы выводятся не в том порядке, в каком мы добавляли их в коллекцию. Ведь HashSet не обещает упорядочивания своих элементов.

Теперь давайте добавим в коллекцию еще один элемент, совпадающий с каким-нибудь из добавленных ранее. Например, так:

```
fishes.add(new Fish("trout",2.2,150));
```

Выведем на экран содержимое коллекции и убедимся, что теперь в коллекции два одинаковых элемента. Но ведь такого быть не должно – HashSet не допускает наличия в коллекции одинаковых элементов. В чем здесь проблема?

Проблема заключается в способе проверки элементов на одинаковость. Они сравниваются по хеш коду. Хеш код получается из метода hashCode(). Поскольку мы не перегрузили в нашем классе этот метод, то при добавлении объектов класса Fish в нашу коллекцию вызывался метод hashCode() из класса Object. А вы знаете, как работает метод hashCode() из класса Object? Он создает хеш код, преобразуя адрес объекта в целое число.

Поэтому он создал для двух одинаковых объектов класса Fish разные хеш коды, так как адреса у этих объектов разные. Чтобы исправить ситуацию, нам надо переопределить в классе Fish метод hashCode(). А если переопределяется hashCode(), то надо переопределить и метод equals(). Давайте сделаем это.

Переопределение этих методов – достаточно интересный процесс сам по себе. Чтобы получить представления о тех задачах, которые надо при этом решить, и рассмотреть способы их решения, можете ознакомиться с материалами по этой ссылке: http://stackoverflow.com/

questions/113511/best-implementation-for-hashcode-method, или с другими подобными материалами.

Давайте добавим в наш класс Fish переопределение этих двух методов:

```
public boolean equals(Object o)
{
    if (o == this) return true;
    if (!(o instanceof Fish))
    {
        return false;
    }

    Fish tmp = (Fish) o;

    return (tmp.name.equals(this.name) &&
        tmp.weight == this.weight &&
        tmp.price == (this.price));
}

public int hashCode()
{
    int code = 17;
    code = 31 * code + this.name.hashCode();
    code = 31 * code + (int)this.weight;
    code = 31 * code + (int)this.price;
    return code;
}
```

У вас возник вопрос, почему используются значения 17 и 31? Отвечаю: потому что это простые числа. Если у вас возникли еще вопросы по этому поводу – читайте материалы по предыдущей ссылке.

Выполните этот код еще раз и убедитесь, что теперь совпадающий элемент в коллекцию не добавился. Все

просто, не так ли? Не спешите отвечать утвердительно. Здесь есть один момент, с которым надо разобраться. Если этого не сделать, то ваше приложение, использующее любую коллекцию-множество, может принести вам головную боль.

Для дальнейшего разговора нам надо вспомнить, что такое immutable и mutable типы в Java. Immutable типы – это типы, объекты которых остаются неизменными после их создания. Другими словами, если вы создали объект immutable типа, то после его создания у вас (и ни у кого другого) нет никакой возможности изменить в созданном объекте какое-либо поле. Каким образом достигается такая неизменяемость – не важно. Например, в классе нет никаких сеттеров, или поля объявлены как final, или даже полей в классе нет совсем. Например, в Java тип String – типичный immutable тип. Соответственно, mutable тип – это тип, состояние объектов которого можно изменять после создания.

Наш класс Fish является immutable, потому что поля в нем private и нет никаких сеттеров, чтобы изменять значения полей после создания объекта. Представим, что мы добавили в класс сеттер для изменения цены. Затем для какого-то объекта класса Fish, добавленного во множество, мы изменили цену. Уже понятно, к чему я веду? Посмотрите на реализацию hashCode() в классе Fish. Если мы изменим price, то изменится хеш код элемента! А это может привести к самым непредвиденным последствиям.

Давайте точнее сформулируем проблему: если реализация метода hashCode() использует mutable поля класса, то поведение, например, метода contains() будет

совершенно неадекватным. Он будет говорить вам, что какого-то элемента нет в коллекции, при том, что этот элемент в коллекции будет.

Например, в официальной документации по этому поводу говорится следующее:

> ***Note:*** *Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.* ***http://docs.oracle.com/javase/7/docs/api/java/util/Set.html.***

Вопросы переопределения методов hashCode() и equals() подробно разбираются в книге Effective Java Джошуа Блока (https://www.amazon.com/Effective-Java-2nd-Joshua-Bloch/dp/0321356683), в разделах 7 и 8.

Вообще говоря, изменение элементов любого множества – задача далеко не тривиальная. Представьте, что у вас есть множество {1, 2, 3}, и вы хотите изменить значение второго элемента на 1, а в множестве не может быть двух одинаковых элементов. Как правило, изменение элемента должно выполняться в два этапа: удаление требуемого элемента и последующая его вставка в коллекцию.

LinkedHashSet

LinkedHashSet содержит все те же методы, что и класс HashSet и отличается от последнего тем, что хранит элементы коллекции в порядке их добавления. Эта коллекция немного медленнее, чем HashSet.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html.

Из особенностей этой коллекции можно отметить такую. Если производится попытка добавить в коллекцию элемент, который в ней уже существует, то возможны два варианта:

1. Совпадающий с добавляемым существующий элемент коллекции удаляется, и в коллекцию добавляется новый елемент;
2. Коллекция не изменяется, и добавляемый элемент отвергается. LinkedHashSet выполняет второй сценарий.

Во всех остальных своих чертах эта коллекция совпадает с рассмотренной коллекцией HashSet.

LinkedHashSet имеет такие конструкторы:

```
LinkedHashSet ()      создает пустую коллекцию;
LinkedHashSet (Collection <? extends E> c)  создает
                      коллекцию, в которую при
                      создании добавляются все
                      элементы коллекции c;

LinkedHashSet (int capacity)      создает коллекцию с
                      начальной емкостью capacity и с
                      фактором загрузки равным 0.75;

LinkedHashSet (int capacity, float loadfactor) создает
                      коллекцию с начальной емкостью
                      capacity и с заданным фактором
                      загрузки.
```

Как видите, здесь тоже полное совпадение с HashSet.

TreeSet

Эта коллекция для хранения своих элементов использует дерево и поэтому всегда содержит элементы в отсортированном по возрастанию порядке. Правда, обработка элементов в TreeSet выполняется медленнее, чем в HashSet или LinkedHashSet.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html.

TreeSet имеет такие конструкторы:

```
TreeSet ()          создает пустую коллекцию;
TreeSet (Collection <? extends E> c)   создает
                    коллекцию, в которую при
                    создании добавляются все элементы
                    коллекции c, сортированные в
                    порядке, естественном для c;
TreeSet (Comparator<? super E> comparator) создает
                    пустую коллекцию с порядком
                    сортировки, заданным в компараторе
                    comparator;
TreeSet(SortedSet<E> s) создает коллекцию,
                    содержащую те же элементы, что и
                    коллекция s, с теми же правилами
                    сортировки.
```

Выполните такой код:

```
TreeSet<String> ts = new TreeSet<>();
ts.add("Georgia");
ts.add("Argentina");
ts.add("Ukraine");
ts.add("Belgium");
ts.add("Canada");
```

```
System.out.println("Collection:"+ts);
System.out.println("Collection's size:"+ts.size());
```

Вы увидите список добавленных стран в отсортированном порядке:

```
Collection:[Argentina, Belgium, Canada, Georgia, Ukraine]
Collection's size:5
```

Попробуем создать коллекцию TreeSet из объектов нашего класса Fish. Если использовать класс Fish в том виде, в каком он у нас сейчас существует, мы получим исключение ClassCastException: listexample.Fish cannot be cast to java.lang.Comparable.

В переводе на человеческий язык это означает, что JVM не знает, как сравнивать объекты нашего класса Fish. По имени? По весу или цене? Чтобы решить эту проблему, надо использовать либо интерфейс Comparable, либо интерфейс Comparator. Каждый из этих интерфейсов позволяет задать правила сравнения объектов какого-либо класса.

Рассмотрим сначала интерфейс Comparable и укажем, что хотим сортировать рыбок по цене. Для этого надо указать, что наш класс Fish реализует интерфейс Comparable. В этом интерфейсе объявлен метод с сигнатурой int compareTo(Object o). Он вызывается от имени тестируемого объекта класса, а в параметре ему передается объект, с которым надо выполнить сравнение. Логика такая: если этот метод возвращает целое значение больше 0, значит, тестируемый объект больше переданного, если метод возвращает 0, значит, оба объекта равны, и если ме-

тод возвращает значение меньше 0 – значит, тестируемый объект меньше переданного. Еще раз обратите внимание, что тип возвращаемого значения – int. Параметр метода compareTo() является объектом типа Object, и его надо приводить к требуемому типу (в нашем случае – к Fish). Однако существует версия параметризированного интерфейса Comparable, при применении которой тип параметра метода compareTo() будет типа вашего класса. Этой версией Comparable мы и воспользуемся. Приведем класс Fish к такому виду:

```
public class Fish implements Comparable<Fish>
{
    private String name;
    private double weight;
    private double price;

    public Fish(String name, double weight, double price)
    {
        this.name=name;
        this.weight=weight;
        this.price=price;
    }

    public boolean equals(Object o)
    {
        if (o == this) return true;
        if (!(o instanceof Fish))
        {
            return false;
        }

        Fish tmp = (Fish) o;

        return (tmp.name.equals(this.name) &&
        tmp.weight == this.weight &&
```

```
        tmp.price == (this.price));
    }

    public int hashCode()
    {
        int code = 17;
        code = 31 * code + this.name.hashCode();
        code = 31 * code + (int)this.weight;
        code = 31 * code + (int)this.price;
        return code;
    }

    @Override
    public String toString()
    {
        return this.name+"  weight:"+this.weight+"
        price:"+this.price;
    }

    @Override
    public int compareTo(Fish o)
    {
        return (int)(this.price * 100 - o.price * 100);
    }
}
```

Умножение на 100 используется для увеличения точности. Без него рыбы с ценами, например, 125.5 и 125.99, рассматривались бы как одинаковые.

Теперь коллекция рыбок будет создана, и они будут отсортированы по цене. Этот код:

```
TreeSet<Fish> fishes = new TreeSet<>();
fishes.add(new Fish("eel",1.5,120));
fishes.add(new Fish("salmon",2.5,180));
```

```
fishes.add(new Fish("carp",3.5,80));
fishes.add(new Fish("trout",2.2,150));
fishes.add(new Fish("trout",2.2,150));

System.out.println("Collection:"+fishes);
System.out.println("Collection's size:"+fishes.size());
```

выведет на экран такое описание коллекции:

```
Collection:[carp  weight:3.5  price:80.0,
eel  weight:1.5  price:120.0,
trout  weight:2.2  price:150.0,
salmon  weight:2.5  price:180.0]
Collection’s size: 4
```

Здесь все просто, однако есть одна оговорка. Мы можем использовать интерфейс Comparable, если у нас есть возможность изменять код требуемого класса, чтобы добавить классу наследование и реализовать в нем метод compareTo(). А такая возможность есть не всегда. Как быть в случае, если мы не можем изменять определение класса? Если класс нам недоступен для редактирования? В этом случае нам поможет интерфейс Comparator.

Рассмотрим использование интерфейса Comparator для решения той же задачи – указать правила сравнения объектов нашего класса. При этом мы не будем вносить никаких изменений в класс, объекты которого будем сравнивать. Создадим рядом с классом Fish такой класс:

```
public class FishComparator implements Comparator<Fish>
{
    @Override
```

```
    public int compare(Fish o1, Fish o2)
    {
        return (int)(o1.getWeight() * 100 -
               o2.getWeight() * 100);
    }
}
```

Имя этого класса произвольно, а главное в нем то, что он наследует интерфейсу Comparator<Fish>. В этом классе метод compare() переопределяется по тем же правилам, что и метод compareTo() из интерфейса Comparable. Но у этого метода уже определены два параметра, поскольку он обращается к сравниваемому классу извне – ведь операция сравнения бинарная. В случае интерфейса Comparable у метода compareTo() был один параметр, поскольку вторым сравниваемым объектом выступал объект, от имени которого вызывался метод compareTo().

В этом случае мы хотим сортировать рыбок по весу. Умножение на 100 требуется для увеличения точности. Без него рыбки с весом 2.2, 2.5 и 2.8 рассматривались бы как одинаковые.

При этом в классе Fish мы ничего не изменяем. Точнее говоря, нам пришлось добавить в этот класс геттер для веса, чтобы в FishComparator получить доступ к private полю weight объектов класса Fish. Наличие геттера является очень легким требованием к классу, ведь в Java все классы должны представлять собой Java bean.

```
public class Fish
{
    private String name;
```

```
private double weight;
private double price;

public Fish(String name, double weight, double price)
{
    this.name=name;
    this.weight=weight;
    this.price=price;
}

public boolean equals(Object o)
{
    if (o == this) return true;
    if (!(o instanceof Fish))
    {
        return false;
    }
    Fish tmp = (Fish) o;
    return (tmp.name.equals(this.name) &&
            tmp.weight == this.weight &&
            tmp.price == (this.price));
}

public int hashCode()
{
    int code = 17;
    code = 31 * code + this.name.hashCode();
    code = 31 * code + (int)this.weight;
    code = 31 * code + (int)this.price;
    return code;
}

public double getWeight()
{
    return this.weight;
}
```

```
    @Override
    public String toString()
    {
        return this.name+"  weight:"+this.weight+"
               price:"+this.price;
    }
}
```

Теперь при создании коллекции TreeSet надо в конструктор коллекции передать объект класса FishComparator.

```
TreeSet<Fish> fishes = new TreeSet<>(new FishComparator());
fishes.add(new Fish("eel",1.5,120));
fishes.add(new Fish("salmon",2.5,180));
fishes.add(new Fish("carp",3.5,80));
fishes.add(new Fish("trout",2.2,150));
fishes.add(new Fish("trout",2.8,150));

System.out.println("Collection:"+fishes);
System.out.println("Collection's size:"+fishes.size());
```

Вот вывод этого кода:

```
Collection:[eel  weight:1.5  price:120.0,
trout  weight:2.2  price:150.0,
salmon  weight:2.5  price:180.0,
trout  weight:2.8  price:150.0,
carp  weight:3.5  price:80.0]
Collection's size:5
```

Как видите, мы добились желаемого результата, не изменяя класс Fish.

Запомните, интерфейсы Comparable и Comparator позволяют указать правила сравнения для объектов любого

класса. При этом использование интерфейса Comparable требует доступа к классу для его изменения. Интерфейс Comparator позволяет задать правила сравнения без изменения целевого класса. Обратите внимание на то, что в обоих этих интерфейсах описано по одному методу. Скоро мы поговорим об этом подробнее.

Queue

Queue (читается, как *"кью"*) является базовым для коллекций, хранящих свои элементы в порядке, определяющем очередность их обработки. У этого интерфейса есть единственный прямой класс наследник – PriorityQueue. Рассмотрим этот класс.

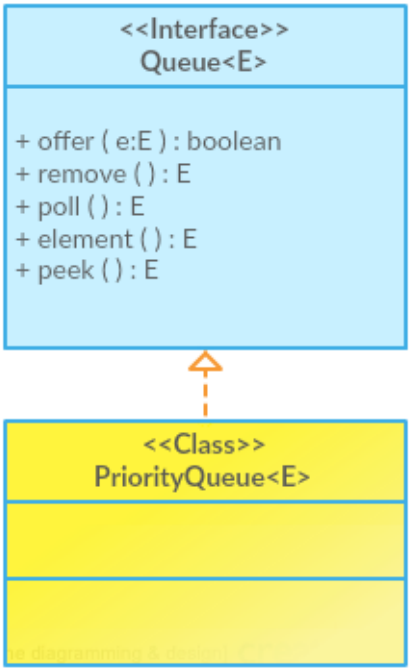

Рис. 4. Интерфейс Queue

PriorityQueue

Коллекция PriorityQueue называется очередью и хранит свои элементы в порядке, необходимом для их обработки. Она упорядочивает элементы либо в порядке их естественной сортировки (задаваемой интерфейсом Comparable), либо в порядке сортировки, определенной в интерфейсе Comparator, переданном в коллекцию через конструктор. По умолчанию очередь реализует принцип FIFO, но это, при необходимости, можно изменить.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html.

PriorityQueue имеет такие конструкторы:

```
PriorityQueue ()    создает пустую коллекцию с
                    естественным порядком размещения;
PriorityQueue (Collection <? extends E> c)
                    создает коллекцию, в которую
                    добавляются все элементы коллекции с
                    и хранятся в естественном для с
                    порядке;
PriorityQueue (int capacity) создает коллекцию с
                    начальной емкостью capacity;
PriorityQueue (int capacity,
                Comparator <? super E> comparator)
                    создает коллекцию с емкостью
                    capacity и с порядком хранения,
                    заданным в компараторе comparator;
PriorityQueue (SortedSet<E> s)
                    создает коллекцию, содержащую те же
                    элементы, что и коллекция s, с тем же
                    порядком хранения
```

```
PriorityQueue (PriorityQueue <E> q)
                    создает коллекцию,  содержащую те же
                    элементы, что и коллекция q, с тем же
                    порядком хранения.
```

У очереди есть одна интересная особенность, связанная с методами для добавления и удаления элементов. Для выполнения каждого из перечисленных действий создано по два метода, и не всегда понятно, чем они отличаются. Поэтому отметьте для себя следующее.

Для добавления в очередь новых элементов можно использовать методы add(value) или offer(value). Разница между ними заключается в том, что при неудачном завершении действия add(value) вызывает исключение, а offer(value) возвращает false.

Для удаления из очереди первого элемента и его возвращения можно использовать методы remove() или poll(). Разница между ними заключается в том, что при неудачном завершении действия remove() вызывает исключение, а poll() возвращает null.

Для извлечения из очереди первого элемента без его удаления можно использовать методы element() или peek(). Разница между ними заключается в том, что при неудачном завершении действия element() вызывает исключение, а peek() возвращает null.

Поскольку значение null является служебным в классе PriorityQueue, вы не можете вставлять в очередь элементы, равные null. Если у вас есть такая потребность, используйте вместо PriorityQueue коллекцию LinkedList.

Рассмотрим использование очереди.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(4);
pq.add(3);
pq.add(1);
pq.offer(9);

System.out.println("Collection:"+pq);
System.out.println("Collection's size:"+pq.size());

System.out.println("\nCollection Using Iterator:");
Iterator<Integer> iter = pq.iterator();

while(iter.hasNext())
{
    System.out.println(iter.next());
}
```

Вывод этого кода будет таким:

```
Collection:[1, 4, 3, 9]
Collection's size:4

Collection Using Iterator:
1
4
3
9
```

Как написано в документации по приведенной выше ссылке:

> *The Iterator provided in method iterator() is not gua ranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray()).*

Именно это мы и видим в приведенном фрагменте кода.

Рассмотрим разные способы получения первого элемента очереди: с удалением этого элемента из очереди и без удаления.

```
System.out.println("Picking the head of the queue: "
       + pq.peek());
System.out.println("Collection:"+pq);
System.out.println("Collection's size:"+pq.size());
System.out.println("Polling the head of the queue: "
       + pq.poll());
System.out.println("Collection:"+pq);
System.out.println("Collection's size:"+pq.size());
```

Вывод этого кода будет таким:

```
Picking the head of the queue: 1
Collection:[1, 4, 3, 9, 12]
Collection's size:5
Polling the head of the queue: 1
Collection:[3, 4, 12, 9]
Collection's size:4
```

Как видите, использование метода peek() позволяет получить первый элемент очереди, не удаляя его, в то время как использование метода poll() удаляет этот элемент из очереди.

В рассмотренных примерах элементы хранились в очереди в порядке их добавления. Давайте создадим очередь с компаратором, элементы которой будут храниться в ней не в порядке добавления в очередь, а в порядке,

определяемом компаратором. Мы работали с очередью с элементами типа Integer, поэтому создадим компаратор для этого типа, только сортировать будем в порядке уменьшения значений элементов.

```
//create the comparator
Comparator<Integer> comparator = new Comparator<Integer>()
{
    @Override
    public int compare(Integer o1, Integer o2)
    {
        if( o1 > o2 )   //if first element is greater
        { //we return negative value
            return -1; //to get descending sort order
        }

        if( o1 < o2 ) //if first element is less
        {             //we return positive value
            return 1; //to get descending sort order
        }

        return 0;     //if the elements are equal
    }
};

//create queue with Integer elements in random order
Queue<Integer> pq = new PriorityQueue<>(comparator);
pq.add(4);
pq.add(3);
pq.add(5);
pq.add(9);
pq.offer(1);

//show the collection
Iterator<Integer> iter = pq.iterator();
while(iter.hasNext())
```

```
{
     System.out.println(iter.next());
}
```

Вывод этого кода будет таким:

```
9
5
4
3
1
```

Как видите, элементы очереди перебираются итератором в соответствии с нашим компаратором. А в какой очередности элементы будут извлекаться из очереди? Проверим. Добавьте в код такой цикл:

```
System.out.println("Removing elements from the queue:");
while( !pq.isEmpty() )
{
    System.out.println( pq.remove() );
}
```

Вывод этого кода будет таким:

```
Removing elements from the queue:
9
5
4
3
1
```

Элементы из очереди извлекаются в порядке, заданном компаратором, а не порядке добавления их в очередь.

4. Интерфейс Map

Интерфейс Map является базовым для коллекций, элементы которых являются не отдельными значениями, а парами "ключ-значение". Ключ играет роль индекса для доступа к конкретной паре. Ключи в коллекции уникальны. Типы ключа и значения могут быть произвольными и отличаться друг от друга. Такие коллекции еще называют отображениями или словарями (*Dictionary*), или ассоциативными массивами. Отметьте, что интерфейс Map является самостоятельным и никак не связан с интерфейсом Collection. Map входит в состав JCF как и интерфейс Collection.

На приведенной ниже UML-диаграмме видно, что интерфейсу Map наследуют четыре класса: HashMap, LinkedHashMap, TreeMap и WeakHashMap. Для наглядности на этой диаграмме не указаны еще некоторые промежуточные интерфейсы и абстрактные классы.

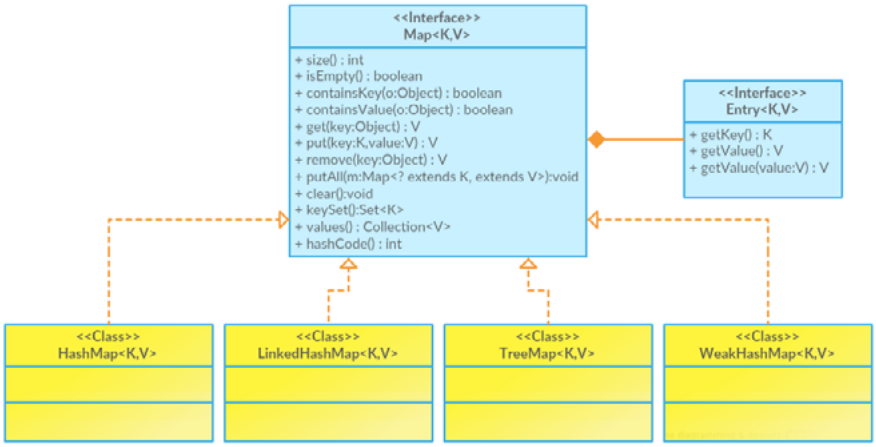

Рис. 5. Интерфейс Map

Рассмотрим эти классы подробнее. Отметьте, что интерфейс Entry определяет внутренний класс в каждом отображении, предназначенный для хранения пар "ключ-значение". При этом экземпляры самого класса Entry хранятся в массиве.

HashMap

HashMap – это неупорядоченная коллекция пар "ключ-значение". Ключу каждой пары сопоставляется хеш код, что ускоряет работу с элементами коллекции. Как ключ, так и значение могут быть равны null. Элементы коллекции хранятся в неупорядоченном виде.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

HashMap имеет такие конструкторы:

```
HashMap ()    создает пустую коллекцию с емкостью 16
              и фактором загрузки 0.75;
HashMap (int capacity) создает коллекцию с начальной
              емкостью capacity и фактором загрузки 0.75;
HashMap (int capacity, float loadfactor) создает
              коллекцию с емкостью capacity и с
              фактором загрузки loadfactor;
HashMap ((Map<? extends K,? extends V> m))
              создает коллекцию,  содержащую те же
              элементы, что и коллекция m.
```

Создадим коллекцию с ключами типа String и значениями типа Integer.

```
HashMap<String, Integer> hm =
                new HashMap<String,Integer>();
```

```
hm.put("Argentina",1);
hm.put("Norway",12);
hm.put("Canada",10);
hm.put("USA",5);
for(Map.Entry m: hm.entrySet())
{
    System.out.println(m.getKey()+" "+m.getValue());
}
```

Вывод этого кода будет таким:

```
Canada 10
Argentina 1
USA 5
Norway 12
```

Как видите, порядок элементов в коллекции НЕ определяется порядком их вставки. Давайте увеличим на 5 значение для элемента с ключом, равным "Argentina".

```
int value = hm.get("Argentina");
hm.put("Argentina", value + 5);
System.out.println("New value of Argentina:
       " + hm.get("Argentina"));
```

Вывод этого кода будет таким:

```
New value of Argentina: 6
```

Теперь создадим отображение с нашим классом Fish.

```
Map<Integer,Fish> map=new HashMap<>();
Fish f1=new Fish("eel",1.5,120);
Fish f2=new Fish("salmon",2.5,180);
```

```
Fish f3=new Fish("carp",2.8,80);
Fish f4=new Fish("trout",2.2,150);

map.put(1,f1);
map.put(2,f2);
map.put(3,f3);
map.put(4,f4);

for(Map.Entry<Integer, Fish> entry:map.entrySet())
{
    int key=entry.getKey();
    Fish b=entry.getValue();
    System.out.println(key+"->"+b);
}
```

Вывод этого кода будет таким:

```
1->eel  weight:1.5  price:120.0
2->salmon  weight:2.5  price:180.0
3->carp  weight:2.8  price:80.0
4->trout  weight:2.2  price:150.0
```

LinkedHashMap

LinkedHashMap расширяет HashMap тем, что создает из элементов (пар "ключ-значение") связный список, хранящий элементы в коллекции в порядке их добавления. Работа с этой коллекцией несколько медленнее, чем с коллекцией HashMap.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html.

LinkedHashMap имеет такие конструкторы:

```
LinkedHashMap ()  создает пустую коллекцию с
                  емкостью 16 и фактором загрузки 0.75;
LinkedHashMap (int capacity) создает коллекцию
                  с начальной емкостью capacity и
                  фактором загрузки 0.75;
LinkedHashMap (int capacity, float loadfactor)
                  создает коллекцию с емкостью
                  capacity и с фактором загрузки
                  loadfactor;
LinkedHashMap (int capacity, float loadfactor,
               boolean accessOrder)
                  создает коллекцию с емкостью
                  capacity и с фактором загрузки
                  loadfactor и порядком доступа к
                  элементам;
                  если accessOrder равен false -
                  элементы в коллекции будут
                  перебираться в порядке их вставки;
                  если accessOrder равен true  -
                  элементы в коллекции будут
                  перебираться в порядке
                  их последней обработки, созданной
                  методами put(), get() и другими;
LinkedHashMap ((Map<? extends K,? extends V> m))
                  создает коллекцию, содержащую те же
                  элементы, что и коллекция m.
```

Рассматривая коллекцию HashMap, мы не рассказали о явлении, называемом коллизией. Поскольку коллизии могут возникать и в LinkedHashMap, поговорим о них здесь.

Что такое коллизия? При добавлении в отображение нового элемента для его ключа вычисляется хеш код, и элемент занимает свое место в коллекции согласно этому хеш коду. Вы должны помнить из предыдущего

раздела нашего урока, что хеш код вычисляется методом hashCode(). Может возникнуть такая ситуация, что у добавляемого элемента хеш код совпадет с хеш кодом другого элемента, уже существующего в коллекции. Это и есть коллизия. Java вполне допускает наличие одинаковых хеш кодов у разных объектов. Ведь мы сами определяем, как должен вычисляться хеш код, переопределяя метод hashCode() в своем классе.

Что происходит при наступлении коллизии? В этом случае в дело вступает метод equals(), который сравнивает два объекта с совпавшими хеш кодами. Если метод equals() сделает вывод, что объекты одинаковы, то существующий объект будет заменен новым. Если метод equals() определит, что объекты разные, то value (из пары *"key-value"*) нового объекта, добавится к уже существующему value объекта с таким же хеш кодом. Вы спросите, а как два значения могут размещаться в одном элементе? Ответ звучит так: для таких value создается связный список, и все они размещаются с этом списке друг за другом. Так обрабатываются коллизии.

К чему это приводит? Конечно же, к замедлению работы с отображением. Поэтому желательно коллизий не допускать. А для этого надо аккуратно переопределять методы hashCode() и equals() в своих классах, которые вы планируете использовать как ключи.

Надо сказать, что начиная с Java 8, при возникновении коллизий вместо списка используется дерево, что несколько уменьшает эффект замедления работы.

Давайте намеренно создадим коллизию. Для этого используем в качестве ключа объекты нашего класса

Fish. Но при этом изменим в этом классе переопределение метода hashCode(), чтобы получать одинаковые хеш коды для разных объектов класса Fish. Приведем класс Fish к такому виду:

```
public class Fish
{
    private String name;
    private double weight;
    private double price;

    public Fish(String name, double weight, double price)
    {
        this.name=name;
        this.weight=weight;
        this.price=price;
    }

    public boolean equals(Object o)
    {
        if (o == this) return true;
        if (!(o instanceof Fish))
        {
            return false;
        }

        Fish tmp = (Fish) o;

        return (tmp.name.equals(this.name) &&
                tmp.weight == this.weight &&
                tmp.price == (this.price));
    }

    public int hashCode()
    {
        return this.name.hashCode();
    }
```

```
    public double getWeight()
    {
        return this.weight;
    }

    @Override
    public String toString() {
        return this.name+"  weight:"+this.weight+"
               price:"+this.price;
    }
}
```

Обратите внимание, что сейчас хеш код для объектов нашего класса вычисляется только на основании текстового значения name. При этом метод equals() по-прежнему считает объекты равными только в случае совпадения значений всех трех полей. Создадим коллекцию с рыбными ключами, при этом из пяти рыбин у нас будет три лосося (с одинаковыми хеш кодами). К тому же объекты f4 и f5 у нас будут одинаковыми с точки зрения метода equals(). Мы попробуем добавить эта два одинаковых объекта в коллекцию с разными значениями.

```
Map<Fish,Integer> map=new LinkedHashMap<>();
Fish f1=new Fish("eel",1.5,120);
Fish f2=new Fish("salmon",2.5,180);
Fish f3=new Fish("salmon",3.2,220);
Fish f4=new Fish("salmon",2.2,150);
Fish f5=new Fish("salmon",2.2,150);

map.put(f1,120);
map.put(f2,180);
map.put(f3,220);
map.put(f4,150);
```

```
map.put(f5,1000);

for(Map.Entry<Fish, Integer> entry:map.entrySet())
{
    Fish key=entry.getKey();
    int b=entry.getValue();
    System.out.println(key+"->"+b);
}
```

Посмотрим, что выведет этот код, и подробно обсудим полученный результат.

```
eel  weight:1.5  price:120.0->120
salmon  weight:2.5  price:180.0->180
salmon  weight:3.2  price:220.0->220
salmon  weight:2.2  price:150.0->1000
```

Первое, что мы видим: в коллекции четыре объекта. Поскольку у объектов f4 и f5 хеш коды одинаковые и метод equals() сообщил, что оба объекта тоже одинаковые, произошло замещение. Объект f5 заменил в коллекции объект f4 (мы не видим в коллекции элемента со значением 150, которое было у f4). Кроме этого, мы увидели все три других элемента с одинаковыми ключами и понимаем, что коллизия произошла, и наша коллекция сейчас содержит под одним хеш кодом сразу три элемента, объединенных в список. У меня работает Java 8, но я говорю о списке, а не о дереве. Почему? Потому что в дерево такие элементы начнут собираться, когда их количество станет больше 8. Что означает тот факт, что в коллекции произошла коллизия? То, что такая коллекция будет работать несколько медленнее,

в сравнении со случаем, когда коллизии нет. Это еще раз подчеркивает важность правильного переопределения методов hashCode() и equals().

TreeMap

Элементами этой коллекции также являются пары "ключ-значение". TreeMap использует для хранения своих элементов дерево, поэтому элементы хранятся в отсортированном порядке по возрастанию. Сортировка происходит в естественном порядке или на основе заданного компаратора. Время доступа к элементам TreeMap мало, поэтому эта коллекция является очень хорошим выбором в том случае, когда вам часто надо находить и выбирать из коллекции требуемые элементы.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html.

TreeMap имеет такие конструкторы:

```
TreeMap ()      создает пустую коллекцию,
                сортированную в естественном порядке;
TreeMap (Comparator<? super K> comparator)
                создает пустую коллекцию с порядком
                сортировки, заданным в компараторе
                comparator;
TreeMap (Map<? extends K,? extends V> m)
                создает коллекцию,  содержащую те же
                элементы, что и коллекция m
                с естественным порядком сортировки;
TreeMap (SortedMap<K,? extends V> s)
                создает коллекцию,  содержащую те же
                элементы, что и коллекция s с тем же
                порядком сортировки.
```

Создадим коллекцию:

```
Map tm = new TreeMap<>();
tm.put("Brazil", "Brazilia");
tm.put("Canada", "Ottawa");
tm.put("Denmark", "Copenhagen");
tm.put("France", "Paris");
tm.put("Ukraine", "Kyiv");
tm.put("Sweden", "Stockholm");

for(Map.Entry e : tm.entrySet())
{
    System.out.println(e.getKey()+" "+ e.getValue());
}
```

Вывод этого кода будет таким:

```
Brazil Brazilia
Canada Ottawa
Denmark  Copenhagen
France Paris
Sweden  Stockholm
Ukraine Kyiv
```

Видно, что элементы отсортированы по правилам сортировки для типа String. Такой порядок сортировки называется естественным.

Для перебора элементов коллекции можно было использовать итератор:

```
Iterator iter = tm.entrySet().iterator();
while(iter.hasNext())
{
    Entry entry = (Entry) iter.next();
```

```
    String key = (String) entry.getKey();
                 //get key of the current element
    String value = (String) entry.getValue();
                 // get value of the current element
    System.out.println(key+"-> "+ value);
}
```

Иногда бывает полезно из какой-либо коллекции Map получить отдельно коллекцию либо ключей, либо значений. Это можно сделать таким образом:

```
List keyList = new ArrayList(tm.keySet());
//list of keys
List valueList = new ArrayList(tm.valueSet());
//list of values
```

В первом случае мы создаем ArrayList из множества ключей отображения tm, полученного методом keySet(). Во втором случае мы создаем ArrayList из множества значений отображения tm, полученного методом valueSet().

Теперь создадим множество с порядком сортировки, заданным в компараторе. В отличие от предыдущего случая, когда мы создавали компаратор в виде отдельного класса, сейчас используем анонимный компаратор. Сделаем ключами объекты нашего класса Fish и будем сортировать эти объекты по цене. С объектами f4 и f5 снова произойдет коллизия.

```
Map<Fish,Double> tmc = new TreeMap<>(new
                       Comparator<Fish>(){
    @Override
    public int compare(Fish o1, Fish o2) {
```

```
        return (int)(o1.getPrice() * 100 -
                o2.getPrice() * 100);
    }
});

Fish f1=new Fish("eel",1.5,120);
Fish f2=new Fish("salmon",2.5,180);
Fish f3=new Fish("trout",3.2,220);
Fish f4=new Fish("salmon",2.2,150);
Fish f5=new Fish("salmon",2.2,150);

tmc.put(f1, 120.0);
tmc.put(f2, 180.0);
tmc.put(f3, 220.0);
tmc.put(f4, 150.0);
tmc.put(f5, 150.0);

for(Object eo : tmc.entrySet())
{
    Map.Entry e=(Map.Entry)eo;
    System.out.println(e.getKey()+"->"+ e.getValue());
}
```

Вывод этого кода будет таким:

```
eel weight:1.5 price:120.0->120.0
salmon weight:2.2 price:150.0->150.0
salmon weight:2.5 price:180.0->180.0
trout weight:3.2 price:220.0->220.0
```

WeakHashMap

WeakHashMap использует для обращения к значениям своих элементов так называемые "слабые ссылки" (*weak reference*). Эти ссылки особым образом обрабатываются

системой сборки мусора. Как только ключ какого-либо элемента коллекции перестает использоваться вашим кодом, этот элемент удаляется системой сборки мусора. Другими словами, если ваш код каким-либо образом удалил ключ какого-то элемента такой коллекции, то при следующем запуске сборки мусора этот элемент будет удален из коллекции. Это приведет к "странным" последствиям, таким как уменьшение размера коллекции. Коллекция такого вида полезна, если для вас важно экономить ресурсы, если вам необходимо хранить очень большие объемы данных. Другим полезным качеством такой коллекции является отсутствие утечек памяти, которые обычно создаются "потерянными" ссылками. В случае WeakHashMap потерянные ссылки удаляются сборщиком мусора, исключая утечку памяти.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html.

WeakHashMap имеет такие конструкторы:

```
WeakHashMap () создает пустую коллекцию с
               емкостью 16 и фактором загрузки 0.75;
WeakHashMap (int capacity)
               создает коллекцию с начальной
               емкостью capacity и фактором
               загрузки 0.75;
WeakHashMap (int capacity, float loadfactor)
               создает коллекцию с емкостью capacity
               и с фактором загрузки loadfactor;
WeakHashMap ((Map<? extends K,? extends V> m))
               создает коллекцию,  содержащую те же
               элементы, что и коллекция m.
```

Посмотрим, как это работает.

```
Map<Fish,Double> wtm = new WeakHashMap<>();
Fish f1=new Fish("eel",1.5,120);
Fish f2=new Fish("salmon",2.5,180);
Fish f3=new Fish("trout",3.2,220);

wtm.put(f1, 120.0);
wtm.put(f2, 180.0);
wtm.put(f3, 220.0);

System.out.println("Before:");
for(Object eo : wtm.entrySet())
{
    Map.Entry e=(Map.Entry)eo;
    System.out.println(e.getKey()+"->"+ e.getValue());
}
f2=null;
//do something huge to invoke garbage collector
for(int i=0; i<10000; i++) {
    byte b[] = new byte[1000000];
    b = null;
}

System.out.println("After:");
for(Object eo : wtm.entrySet())
{
    Map.Entry e=(Map.Entry)eo;
    System.out.println(e.getKey()+"->"+ e.getValue());
}
```

Вывод этого кода будет таким:

```
Before:
salmon  weight:2.5  price:180.0->180.0
trout  weight:3.2  price:220.0->220.0
eel  weight:1.5  price:120.0->120.0
```

```
After:
trout  weight:3.2  price:220.0->220.0
eel  weight:1.5  price:120.0->120.0
```

Мы создали коллекцию из трех элементов и увидели их после выполнения первого цикла. Затем удалили объект, который является ключом одного из элементов коллекции. После этого выполнили бессмысленные действия, чтобы загрузить память и вызвать сборку мусора. После чего снова вывели содержимое коллекции на экран и увидели, что в ней осталось лишь два элемента. Элемент коллекции, ключ которого "потерял" свою ссылку, был автоматически удален из коллекции. Замените в этом примере WeakHashMap на другую коллекцию, например, HashMap, и убедитесь, что удаления элемента из коллекции не произойдет.

5. Синхронизация коллекций

Все рассмотренные выше коллекции не являются синхронизированными, а следовательно, не являются потокобезопасными. Если выполнять доступ к коллекции одновременно из нескольких потоков, это может привести к исключительной ситуации. Почему коллекции созданы не синхронизированными? В ранних версиях Java присутствовали синхронизированные коллекции. Однако оказалось, что такие синхронизированные коллекции демонстрируют намного меньшую производительность, чем их не синхронизированные версии. За синхронизацию надо платить. Поэтому разработчики Java создали коллекции изначально не синхронизированными, предупредив нас о необходимости выполнять синхронизацию самостоятельно, когда это необходимо.

Кроме этого, разработчики коллекций предоставили нам механизм превращения не синхронизированной коллекции в синхронизированную, чтобы избавить нас от проблем самостоятельного выбора способа синхронизации и выполнения этой синхронизации.

Вы уже встречались с классом Collections, в котором реализованы методы, называемые алгоритмами. В этом классе есть набор методов с именами:

```
Collections.synchronizedList()
Collections.synchronizedMap()
Collections.synchronizedSet()
Collections.synchronizedSortedList()
Collections.synchronizedSortedSet()
```

Каждый из этих методов позволяет превратить обычную коллекцию в синхронизированную. Например:

```
Map<Integer, String> unsafeHm = new HashMap<>();
Map<Integer, String> safeHm = Collections.
             synchronizedMap(unsafeHm);
```

или:

```
List<String> safeList = Collections.
             synchronizedList(new ArrayList<>());
```

Понятно, что синхронизированная таким образом коллекция будет проигрывать в производительности своей не синхронизированной версии.

При этом отметьте такую особенность: итератор синхронизированной коллекции не является синхронизированным! Рассмотрим такой код:

```
ArrayList<String> sal = Collections.
                  synchronizedList(new ArrayList<>());
sal.add("Argentina");
sal.add("Bulgaria");
sal.add("Canada");
sal.add("Denmark");
sal.add("Narnia");

System.out.println("\nCollection Using Iterator:");

Iterator<String> iter = sal.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

Опасность этого фрагмента кода располагается в цикле итератора. Если при выполнении этого цикла в одном потоке в другом потоке коллекция будет изменяться, то возникнет исключительная ситуация Concurrent ModificationException. Поэтому при использовании синхронизированных таким образом коллекций ***не забывайте отдельно синхронизировать блоки, использующие итераторы***. Чтобы наш код был потокобезопасным, он должен выглядеть так:

```
synchronized (sal) {
    Iterator<String> iter = sal.iterator();
    while (iter.hasNext()) {
}
```

6. Concurrent коллекции

Из предыдущего раздела вы должны понять, что все методы Collections.synchronizedXXX() в качестве объекта синхронизации используют саму коллекцию, т.е. блокируют всю коллекцию целиком. Другими словами, если один поток работает с такой коллекцией, то все остальные потоки, которым тоже надо работать с ней, блокируются и ждут завершения работы текущего потока. Чтобы избежать такого замедления при использовании синхронизированных коллекций, в Java 5 появились concurrent collections. Эти коллекции собраны в пакете java.util.concurrent.

Еще раз отметьте для себя разницу между синхронизированной коллекцией и concurrent коллекцией. Синхронизированная коллекция работает медленно, потому что в качестве объекта синхронизации используется вся коллекция целиком. Concurrent коллекция не блокируется полностью, а использует более тонкие способы синхронизации. В пакете java.util.concurrent собраны не только concurrent коллекции, а еще и ряд инструментов, делающих работу с этими коллекциями более эффективной. Этот пакет достаточно сложен и не является темой рассмотрения нашего урока. Однако для примера мы рассмотрим использование одной из concurrent коллекций.

Рассмотрим использование класса CopyOnWrite ArrayList<E>. Эта коллекция допускает одновременное чтение из коллекции многими потоками или одновременную запись в коллекцию одним потоком и при этом

чтение этой же коллекции другими потоками. Это достигается за счет того, что для методов, выполняющих запись (add(), set(), remove() и других), создается копия элементов и за счет этого достигается такой эффект, что все операции чтения выполняются с собственными копиями.

Создадим коллекцию CopyOnWriteArrayList и два потока. Один поток будет выполнять запись в нашу коллекцию, другой – читать из нее.

Поток, выполняющий запись:

```
public class MyWriter extends Thread
{
    private List<String> list;
    private int item;

    public MyWriter(String name, List<String> list)
    {
        this.list = list;
        item=0;
        super.setName(name);
    }

    public void run()
    {
        while (true)
        {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            String new_item="element"+item++;
            list.add((String)(new_item));
```

```
            System.out.println(super.getName() +
                "New element added!");
    }
}
```

Этот поток работает постоянно, записывая в коллекцию новые элементы. Новые элементы создаются в строке new_item как конкатенация слова "*element*" и постоянно увеличивающегося целого значения. После добавления каждого элемента в консоль выводится сообщение *"New element added!"*. Для имитации выполнения сложной работы поток приостанавливается на одну секунду при записи каждого нового элемента.

Поток, выполняющий чтение:

```
public class MyReader extends Thread
{
    private List<String> list;

    public MyReader(String name, List<String> list) {
        this.list = list;
        super.setName(name);
    }

    public void run() {
        while (true)
        {
            String info = super.getName() + ":";
            Iterator<String> iterator = list.iterator();
            while (iterator.hasNext())
            {
                String el = iterator.next();
                info += " " + el;
                try {
```

```
                    Thread.sleep(10);
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
            System.out.println(info);
        }
    }
}
```

Задача этого потока заключается в том, что он должен собрать в строку info все элементы коллекции и вывести их в консольное окно. Элементы в строке info отделяются один от другого пробелами. Смысл этой работы в том, что, пока будет выполняться чтение коллекции (перебор ее элементов с помощью итератора), поток MyWriter будет добавлять в нее новые элементы. И по мере выполнения приложения MyReader будет выводить в консольное окно изменяющуюся коллекцию с новыми элементами.

Осталось добавить простую реализацию метода main:

```
public static void main(String[] args)
{
    List<String> cowal = new CopyOnWriteArrayList<>();

    cowal.add("Belgium");
    cowal.add("USA");
    cowal.add("Poland");
    cowal.add("Brazil");
    cowal.add("Canada");

    Thread twriter=new MyWriter("MyWriter",cowal);
    twriter.start();
    Thread treader=new MyReader("MyReader",cowal);
```

```
    treader.start();
}
```

Запустите приложение. Оно будет выполняться бесконечно, поэтому прервите его выполнение, когда получите убедительный вывод. Вывод этого кода будет таким:

```
MyReader: Belgium USA Poland Brazil Canada
MyReader: Belgium USA Poland Brazil Canada
MyReader: Belgium USA Poland Brazil Canada
MyWriterNew element added!
MyReader: Belgium USA Poland Brazil Canada
MyReader: Belgium USA Poland Brazil Canada element0
MyReader: Belgium USA Poland Brazil Canada element0
MyWriterNew element added!
MyReader: Belgium USA Poland Brazil Canada element0
MyReader: Belgium USA Poland Brazil Canada element0
          element1
MyReader: Belgium USA Poland Brazil Canada element0
          element1
MyWriterNew element added!
```

Я удалил некоторое количество повторяющихся строк. Думаю, все это выглядит достаточно убедительно. Мы одновременно выполняли запись в коллекцию в одном потоке и ее перебор в другом потоке и не получили никаких исключительных ситуаций.

Кроме рассмотренного класса в пакете java.util.concurrent есть еще много других коллекций. Таких как:

```
CopyOnWriteArraySet<E>
ConcurrentMap<K,V>
ConcurrentHashMap<K,V>
```

```
ConcurrentNavigableMap<K,V>
ConcurrentSkipListMap<K,V>
ConcurrentSkipListSet<K,V>
```

По именам понятно, что каждая из этих коллекций является усовершенствованием соответствующей не синхронизированной коллекции. Кроме этих коллекций в пакете есть еще много классов для создания concurrent очередей и списков.

Вывод: *concurrent коллекции обеспечивают эффективную синхронизацию при многопоточном доступе. Работают эти коллекции быстрее, чем их аналоги, синхронизированные с помощью методов Collections.synchronizedXXX(). Но работа с ними требует от разработчика определенных усилий.*

7. Аннотации

Аннотации в Java – это механизм, позволяющий добавлять в созданный код, некую дополнительную информацию, называемую метаданными. Эта информация может использоваться на этапе компиляции кода, на этапе сборки приложения или на этапе выполнения. Аннотации появились в Java начиная с версии Java 5. Есть аннотации, встроенные в язык Java его разработчиками. Их мы рассмотрим позже. А еще есть механизм, позволяющий программисту создавать свои собственные аннотации.

В краткой форме описания аннотация выглядит так:

```
@AnnotationName
```

где AnnotationName – имя аннотации.

Аннотация также может содержать атрибуты, тогда она будет выглядеть так:

```
@ AnnotationName (attribute1 = "value1",
                  attribute2 = "value2")
```

где attribute1 и attribute2 – имена атрибутов, а value1 и value2 – значения атрибутов.

Если у аннотации только один атрибут, то принято называть его value:

```
@ AnnotationName (value = "value1")
```

Предыдущую аннотацию также можно записать в еще более краткой форме:

```
@ AnnotationName ("value1")
```

Аннотации могут размещаться перед определением класса, метода, поля, параметра метода или локальной переменной.

Встроенные в Java аннотации

Мы уже отметили, что есть аннотации, встроенные в язык Java его разработчиками. Это, например, такие аннотации:

```
@Deprecated
@Override
@SuppressWarnings
```

Все они используются на этапе компиляции.

- ***@Deprecated*** говорит о том, что объект, перед которым она расположена уже считается устаревшим и скоро будет удален из реализации языка. Во время компиляции, компилятор будет выдавать предупреждающие сообщения о каждом случае использования устаревшего объекта.
- ***@Override*** используется в классе наследнике перед переопределяемым методом базового класса. Данный атрибут не является обязательным и переопределение метода можно выполнять и без него. Однако, рекомендуется всегда указывать этот атрибут перед

переопределяемыми методами, чтобы человек, работающий с классом, понимал, что такие методы определены в базовом классе, и при их переопределении требуется сохранять соответствие сигнатуры. Если вы не укажете эту аннотацию при переопределении методов базового класса, компилятор выведет вам предупреждение о том, что метод в классе наследнике не переопределяет метод базового класса.

- ***@SuppressWarnings*** используется для того, чтобы подавлять предупреждения компилятора, например, об использовании небезопасного преобразования типов, об использовании устаревших методов и т.п.

Пользовательские аннотации

Пользователь может создавать свои собственные аннотации. Рассмотрим этот процесс. Аннотация создается в собственном отдельном файле, как класс или интерфейс. Определяется аннотация таким образом:

```
@interface CodeAuthor {
    String name();
    int version();
    String edited();
    String[] asisstants();
}
```

Такое определение создает аннотацию с именем Code Author с четырьмя атрибутами, имена и типы которых описаны в определении. При использовании эта аннотация может выглядеть так:

```
@interface CodeAuthor ( name = "Indiana Jones",
           version = "1", edited = "20/04/2017",
           asisstants = { "Marion", "Sallah"}
)

class LostArk
{
    //class members
}
```

Если вы не хотите указывать значения для каждого из атрибутов, тогда вы должны при определении указать для такого атрибута значение по умолчанию. Например, так:

```
@interface CodeAuthor {
    String name();
    int version() default "1";
    String edited() default "01/01/1970";
    String[] asisstants();
}
```

Теперь атрибуты version и edited можно опускать:

```
@interface CodeAuthor ( name = "Indiana Jones",
           asisstants = { "Marion", "Sallah"}
)
class LostArk
{
    //class members
}
```

Создавая свою аннотацию, вы можете указать к каким элементам она должна применяться. Если бы мы хотели указать при определении нашей аннотации, что

она должна применяться только к типам (классам, интерфейсам, перечислениям), нам надо было бы добавить в определении такую строку:

```
@Target({ElementType.TYPE})
@interface CodeAuthor {
    String name();
    int version() default "1";
    String edited() default "01/01/1970";
    String[] asisstants();
}
```

Класс ElementType содержит такие допустимые значения для области применимости аннотации:

```
ElementType.ANNOTATION_TYPE
ElementType.CONSTRUCTOR
ElementType.FIELD
ElementType.LOCAL_VARIABLE
ElementType.METHOD
ElementType.PACKAGE
ElementType.PARAMETER
ElementType.TYPE
```

Если при определении своей аннотации указать еще аннотацию @Inherited, то созданная вами аннотация будет передаваться и классам наследникам, если таковые будут у вашего класса.

```
@Inherited
@Target({ElementType.TYPE})
@interface CodeAuthor {
    String   name();
    int   version() default "1";
```

```
    String edited() default "01/01/1970";
    String[] asisstants();
}
```

Аннотация @Documented предназначена для инструмента JavaDoc. Если вы будете создавать документацию для своего кода с помощью JavaDoc, то ваша аннотация, объявленная с @Documented, будет включена в сгенерированную документацию. В завершение этого раздела приведем пример класса с разными аннотациями: для полей класса, для методов, для параметров:

```
@Entity
public class Picture {
    @PrimaryKey
    protected Integer pictureId;

    @Persistent
    protected String pictureName = null;

    @Getter
    public String getPictureName() {
        return this. pictureName;
    }

    @Setter
    public void setPictureName(@Optional pictureName) {
        this. pictureName = pictureName;
    }
}
```

8. Анонимные классы

Анонимные классы дают возможность определять новый класс и создавать его экземпляр одновременно. Это похоже на локальные вложенные классы, но у анонимных классов нет имен. Использование анонимных классов делает код более компактным. Код, объявляющий анонимный класс и создающий его объект по своей сути является выражением. Анонимный класс создается, как производный от какого-то базового типа, описанного в абстрактном классе или в интерфейсе. Такой класс удобно использовать, когда вам требуется единичный объект базового типа и для создания этого объекта, использование классического определения класса будет избыточным. Но еще полезнее использование анонимных классов в тех случаях, когда вам требуется переопределить какие-либо методы при создании объекта.

Рассмотрим подробнее выражение для создания анонимного класса. В таком выражении всегда присутствует инструкция new.

Создадим в интерфейсе базовый тип, для которого потом будем создавать объекты в анонимных классах.

```
public interface Group {
    String bestAlbum();
}
```

Теперь покажем, как можно создать анонимный класс, а следовательно, и объект этого класса, производный от заданного интерфейса. Например, в методе main() можно поступить так:

```
public static void main(String[] args)
{
    Group pinkFloyd = new Group(){
        @Override
        public String bestAlbum() {
            return "Wish You Were Here";
        }
    };
    String album = pinkFloyd.bestAlbum();
    System.out.println(album);
}
```

Выделенный фрагмент кода демонстрирует создание объекта анонимного класса и одновременную реализацию метода bestAlbum() из базового типа Group. Посмотрите внимательно на этот код. Мы создаем ссылку типа базового интерфейса (Group) и инициализируем ее анонимным (без имени) объектом. Здесь же, при создании этого объекта, мы реализуем метод, так, как нам надо в данном конкретном случае.

Вывод этого кода будет таким:

```
Wish You Were Here
```

Объекты анонимных классов удобно использовать, как параметры методов. Добавим рядом с методом main() еще один метод. Параметр этого метода будет типа нашего базового интерфейса. И от имени этого параметра будет вызываться метод bestAlbum():

```
public static void showGroup(Group group)
{
    System.out.println(group.bestAlbum());
}
```

Вы понимаете, что мы можем вызвать этот метод, передав ему в качестве параметра наш объект pinkFloyd:

```
showGroup(pinkFloyd);
```

Но если мы хотим вызвать его для другой группы, то можем поступить так:

```
showGroup(new Group(){
    @Override
    public String bestAlbum()
    {
        return "A Night At The Opera";
    }
});
```

И снова, выделенный фрагмент кода – это создание анонимного объекта с одновременным переопределением метода bestAlbum().

Вывод этого кода будет таким:

```
A Night At The Opera
```

В качестве базового типа для анонимного класса часто используются встроенные в Java системные интерфейсы. Например, ранее в нашем уроке вы видели примеры создания анонимного класса для интерфейса Comparator.

Используя анонимные классы, надо учитывать некоторые, присущие им ограничения:

- В анонимном классе нельзя создавать static члены;
- Анонимные классы не могут использоваться со спецификаторами доступа public, private, protected, или static;

- Интерфейсы не могут быть анонимными, поскольку их нельзя будет реализовать без имени;
- Не следует использовать анонимные классы, если вам надо создать несколько объектов базового класса, поскольку выполнение анонимного выражения в этом случае будет неэффективным;
- В анонимном классе нельзя создавать конструктор.

9. Lambda выражения

Лямбда-выражения – одно из самых ожидаемых нововведений языка, появившееся в версии Java 8. Что же это такое и почему многие разработчики так ожидали появления этой новинки? С чисто синтаксической точки зрения, лямбда-выражения – это просто сокращенный способ записывать выражения или даже целые блоки кода.

Общая структура лямбда-выражения такая:

```
(параметры) -> выражение
```

Или

```
(параметры) -> { выражение1; выражение2; …; выражениеN; }
```

Если справа от –> располагается только одно выражение, оно выполняется и, возможно, возвращает результат. Если надо использовать несколько выражений, они заключаются в {} и выполняются, как тело метода, возможно, возвращая что-нибудь или ничего не возвращая. Если параметры не требуются, слева от –> надо указывать пустые скобки (). Очень часто тип параметров лямбда-выражения можно не указывать, он определяется автоматически по контексту. Можно рассматривать лямбда-выражения, как анонимные методы, оформленные в виде выражения. Давайте разберемся в том, что нового появилось в Java с лямбда-выражениями и рассмотрим основные случаи их использования.

Функциональные интерфейсы

Вспомните, как вы обрабатываете события в библиотеке Swing. Чтобы обработать какое–либо событие, вы должны реализовать интерфейс, в котором описаны методы – обработчики этого события. При этом вы реализуете эти методы, программируя в них, необходимое вам действие. Очень часто такие интерфейсы содержат описание только одного метода. В Java такие интерфейсы играют особую роль и даже получили собирательное название – функциональные интерфейсы. Подробнее почитать об этих интерфейсах можно здесь: https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html.

Типичный представитель функциональных интерфейсов – ActionListener с методом actionPerformed(). Ниже приведен стандартный код по обработке события:

```
JButton btn = new JButton("Test Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button is cliced!");
    }
});
```

Теперь внимательно послушайте. Если вы можете составить лямбда-выражение с параметром, тип которого совпадает с типом параметра метода, описанного в функциональном интерфейсе, то вы можете описать реализацию такого интерфейса с помощью лямбда-выражения. Чтобы сделать это, вы должны передать та-

кое лямбда-выражение в качестве параметра методу addActionListener():

```
btn.addActionListener((e) -> {
    System.out.println("Lambda expressions performs
                        the click");
});
```

Обратите внимание, что сам параметр мы совсем не используем, но он должен присутствовать в лямбда-выражении, поскольку он определен в методе actionPerformed().

Какие еще функциональные интерфейсы приходят вам на ум? Что скажете по поводу Runnable?

Вот стандартный способ создания и запуска нового потока:

```
new Thread(new Runnable() {
    @Override public void run() {
        System.out.println("Common thread creation!");
    }
}).start();
```

А вот с лямбда-выражением:

```
new Thread( () ->
System.out.println("Using lambda expression for
                    thread creation!") ).start();
```

Обратите внимание, у метода run() в интерфейсе Runnable параметров нет, поэтому, в созданном лямбда-выражении их нет тоже.

А кто помнит, сколько методов описано в интерфейсе Comparator? Рассмотрим еще раз классический

способ сортировки коллекции класса Fish, например, по весу:

```
List<Fish> fishes = new ArrayList<>();
fishes.add(new Fish("eel",1.5,120));
fishes.add(new Fish("salmon",2.5,180));
fishes.add(new Fish("carp",3.5,80));
fishes.add(new Fish("tuna",4.2,320));
fishes.add(new Fish("trout",2.8,150));

System.out.println("Before Sorting:");
    for (Fish f : fishes) {
       System.out.println(f);
  }

//sort by weight
Collections.sort(fishes, new Comparator<Fish>() {
    @Override
    public int compare(Fish f1, Fish f2) {
        return (int)f1.getWeight() * 100 -
               (int)f2.getWeight() * 100;
    }
});

System.out.println("After Sorting:");
for (Fish f : fishes) {
    System.out.println(f);
}
```

Мы воспользовались интерфейсом Comparator. С помощью ***анонимного класса*** переопределили единственный метод compare() так, чтобы он сравнивал объекты класса Fish по значению поля weight. Использование анонимного класса позволило нам выполнить переопределение метода compare() на лету, прямо при вызове метода

Collections.sort(). А что, если бы нам понадобилось отсортировать объекты этой коллекции по полю price? Думаю, ответ вам понятен. Надо было бы переопределить метод compare(), чтобы он работал с getPrice(). А что, если нам надо сортировать и по возрастанию и по убыванию? А что, если вместе с этим нам еще понадобится сортировка по имени? Вы понимаете, к чему я клоню?

Воспользуемся преимуществами, которые нам предоставляет Java 8. Одним из них является то, что теперь метод sort() располагается в самом интерфейсе List, и нет необходимости использовать Collections.sort():

```
fishes.sort(new Comparator<Fish>() {
    @Override
    public int compare(Fish f1, Fish f2) {
        return (int)f1.getWeight() * 100 -
               (int)f2.getWeight() * 100;
    }
});
```

Результат будет таким же:

```
Before Sorting:
eel  weight:1.5  price:120.0
salmon  weight:2.5  price:180.0
carp  weight:3.5  price:80.0
tuna  weight:4.2  price:320.0
trout  weight:2.8  price:150.0
After Sorting:
eel  weight:1.5  price:120.0
salmon  weight:2.5  price:180.0
trout  weight:2.8  price:150.0
carp  weight:3.5  price:80.0
tuna  weight:4.2  price:320.0
```

Но еще более впечатляющим будет использование лямбда-выражения для реализации функционального интерфейса Comparator:

```
fishes.sort((Fish f1, Fish f2)->(int)f1.getWeight() *
            100 - (int)f2.getWeight() * 100);
```

Попробуем убрать описание типов параметров лямбда-выражения:

```
fishes.sort((f1, f2)->(int)f1.getWeight() *
            100 - (int)f2.getWeight() * 100);
```

Все работает! Параметры типов были автоматически определены по коллекции. Хотите отсортировать по цене? Пожалуйста:

```
fishes.sort((f1, f2)->(int)f1.getPrice() * 100 -
           (int)f2.getPrice() * 100);
```

Вывод этого кода будет таким:

```
Before Sorting:
eel  weight:1.5  price:120.0
salmon  weight:2.5  price:180.0
carp  weight:3.5  price:80.0
tuna  weight:4.2  price:320.0
trout  weight:2.8  price:150.0
After Sorting:
carp  weight:3.5  price:80.0
eel  weight:1.5  price:120.0
trout  weight:2.8  price:150.0
salmon  weight:2.5  price:180.0
tuna  weight:4.2  price:320.0
```

Мы на лету формируем требуемое нам лямбда-выражение и получаем ожидаемое упорядочивание элементов коллекции. Нам не надо для многих вариантов сортировки переопределять метод compare() интерфейса Comparator.

Мы рассмотрели использование лямбда-выражения для реализации функциональных интерфейсов. Идем дальше.

Интерфейс Predicate

Если вам понравились последние примеры сортировки, то вам будет интересно узнать, что их можно еще значительно улучшить. В Java 8 в пакете java.util.function добавлено несколько новых функциональных интерфейсов. Они позволяют добавлять логику в функциональные методы, делая код еще более компактным и более емким в то же время. Рассмотрим интерфейс Predicate. Вы должны помнить значение этого термина со времени изучения STL. В интерфейсе Predicate описан булевский метод test(), который обычно используется для фильтрования. Воспользуемся коллекцией из объектов типа Fish для создания нескольких предикатов. Сейчас мы будем создавать объекты типа Predicate<Fish>, перегружая метод test() с помощью лямбда-выражений. Каждый такой созданный объект предикат будет позволять выбирать из коллекции элементы по своему критерию отбора. Эти объекты будем передавать в качестве параметра в специальный метод, который будет отбирать из коллекции элементы:

```
public static void getByPredicate(List<Fish> fishes,
    Predicate<Fish> p) {
        for (Fish f : fishes) {
            if (p.test(f)) {
```

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

Выбираем рыб, цена которых больше 200:

```
System.out.println("Fishes more expencive than 200:");
getByPredicate(fishes, (f) -> f.getPrice() > 200 );
```

Вывод этого кода будет таким:

```
Fishes more expencive than 200:
tuna  weight:4.2  price:320.0
```

Выбираем рыб, цена которых больше 100 и вес больше 2:

```
System.out.println("Fishes more expencive than 100
                    and heavier than 2:");
getByPredicate(fishes, (f) -> f.getPrice() > 100 &&
                    f.getWeight() > 2);
```

Вывод этого кода будет таким:

```
Fishes more expencive than 100 and heavier than 2:
trout  weight:2.8  price:150.0
salmon  weight:2.5  price:180.0
tuna  weight:4.2  price:320.0
```

Один метод getByPredicate() позволяет выбирать элементы коллекции по совершенно разным критериям отбора, получая эти критерии во входном параметре в виде лямбда-выражения. Критерии можно создавать динамически, ничего не изменяя в коде отбора.

Приведем еще один пример:

```
System.out.println("Fishes with names longer than
                    5 characters:");
getByPredicate(fishes, (f) -> f.getName().length() > 5);
```

Вывод этого кода будет таким:

```
Fishes with names longer than 5 characters:
salmon  weight:2.5  price:180.0
```

Можно сказать, что во всех случаях вызова метода getByPredicate() мы передаем ему в лямбда-выражении поведение, а не данные.

Перебор коллекций

В этом уроке вы уже видели разные способы перебора элементов коллекций. Все они, так или иначе, были циклами. Рассмотрим еще один способ. В Java 8 в интерфейсе List появился метод forEach(). Этот метод, позволяющий перебирать элементы коллекции, в качестве параметра принимает лямбда-выражение. При переборе элементов коллекции он может применять к каждому из них логику, содержащуюся в лямбда-выражении.

Рассмотрим такой код:

```
List<Integer> items = new ArrayList<>();
items.add(11);
items.add(5);
items.add(120);
items.add(85);
items.add(251);
```

```
items.add(199);
items.forEach(item->System.out.println(item));
```

Такой вызов forEach() приведет к тому, что для каждого элемента коллекции items будет вызван метод System.out.println(). Обратите внимание, что мы снова не указываем в лямбда-выражении тип параметра. В результате выполнения этого кода мы увидим в консольном окне все элементы:

```
11
5
120
85
251
199
```

Теперь изменим элементы коллекции. Каждый элемент со значением больше 100 разделим на 10:

```
items.forEach(item->{
    if(item > 100){
        int index = items.indexOf(item);
        items.set(index, item/10);
    }
});
System.out.println("After:");
items.forEach(item->System.out.println(item));
```

Здесь лямбда-выражение более сложное, состоящее из нескольких строк. Результат выполнения предсказуем:

```
After:
11
```

```
5
12
85
25
19
```

Интерфейс Stream<T>

Еще одним важным дополнением в Java 8 является интерфейс Stream<T>, реализующий последовательность элементов, поддерживающую последовательные и агрегационные операции. Этот интерфейс очень эффективно проявляется в использовании с коллекциями. Мы можем легко получить потоковое (в смысле интерфейса Stream<T>) представление любой коллекции, массива или другого источника данных и обрабатывать его методами интерфейса. Методы этого интерфейса делятся на две группы: промежуточные и терминальные. Промежуточные возвращают поток и потому могут вызываться последовательно друг за другом цепочкой (*chaining calls*). Терминальные не возвращают ничего или возвращают не поток и потому являются последними в цепочке вызова. Такие методы, как filter(), map(), sorted() являются промежуточными, а forEach() – терминальным.

Вернемся к нашей коллекции fishes и уменьшим на 10% цену всех рыб, которые дороже 100.

```
System.out.println("Before:");
fishes.forEach(f->System.out.println(f));
fishes.stream()
    .filter(f -> f.getPrice() > 100)
```

```
    .forEach(f -> f.setPrice( f.getPrice()*0.9 ));
System.out.println("After:");
fishes.forEach(f->System.out.println(f));
```

Сначала от имени коллекции мы вызываем метод stream(), который создает потоковое представление коллекции. Затем для этого потока цепочкой вызываем методы filter() и forEach(), оба принимающие лямбда-выражения в качестве параметров. Первый метод выбирает из коллекции элементы по заданному критерию, а второй – применяет к отобранным элементам требуемую обработку (уменьшает цену на 10%).

Вывод этого кода будет таким:

```
Before:
carp  weight:3.5  price:80.0
eel  weight:1.5  price:120.0
trout  weight:2.8  price:150.0
salmon  weight:2.5  price:180.0
tuna  weight:4.2  price:320.0

After:
carp  weight:3.5  price:80.0
eel  weight:1.5  price:108.0
trout  weight:2.8  price:135.0
salmon  weight:2.5  price:162.0
tuna  weight:4.2  price:288.0
```

Поток легко позволяет выбрать из коллекции подколлекцию по произвольному критерию. Выберем из коллекции fishes после переоценки в отдельную коллекцию все элементы с ценой больше 100.

```
List<Fish> selected = fishes.stream()
    .filter(f -> f.getPrice() > 100)
    .collect(Collectors.toList());

System.out.println("After:");
selected.forEach(f->System.out.println(f));
```

Вывод этого кода будет таким:

```
After:
eel  weight:1.5  price:108.0
trout  weight:2.8  price:135.0
salmon  weight:2.5  price:162.0
tuna  weight:4.2  price:288.0
```

А вот примеры агрегационных операций, выполненных с потоковым представлением коллекции. Мы посчитаем количество элементов в коллекции с ценой больше 100 и их общую стоимость:

```
int number = (int) fishes.stream()
    .filter(f -> f.getPrice() > 100)
    .count();

double cost = fishes.stream()
    .filter(f -> f.getPrice() > 100)
    .mapToDouble(f -> f.getPrice())
    .sum();

System.out.println("number="+number+" total cost="+cost);
```

Вывод этого кода будет таким:

```
number=4  total cost=693.0
```

В рассмотренном нами примере используется метод mapToDouble(), являющийся модификацией метода map(). Эти методы позволяют применять какое-либо действие к каждому перебираемому элементу, напоминая этим метод forEach(), но в отличие от терминального forEach(), методы map() возвращают потоковый объект, не прерывая цепочки вызова. Используя этот метод, создадим из нашей коллекции items новую коллекцию, содержащую квадраты значений элементов исходной коллекции:

```
List<Integer> squares = items.stream().map( (i) -> i*i)
             .collect(Collectors.toList());
System.out.println("Squares:");
squares.forEach(s->System.out.println(s));
```

Вывод этого кода будет таким:

```
Squares:
121
25
144
7225
625
361
```

Для создания потока не обязательно использовать коллекцию. Поток можно создать из набора любых объектов методом of().

```
Stream.of("Argentina", "Bulgaria",
          "Canada","Denmark","Ukraine","USA")
    .filter((c)->c.startsWith("U"))
    .forEach(c->System.out.println(c));
```

Здесь приведен пример создания потока из набора строк. Приведем названия стран в этой коллекции в верхний регистр, с помощью метода map():

```
Stream.of("Argentina", "Bulgaria", "Canada",
          "Denmark", "Ukraine","USA")
    .map(String::toUpperCase)
    .forEach((c)->System.out.println(c));
```

Вывод этого кода будет таким:

```
ARGENTINA
BULGARIA
CANADA
DENMARK
UKRAINE
USA
```

В этом примере вы видите еще одну новинку Java 8 – оператор ::, называемый в официальной документации Method Reference, т.е. ссылка на метод. Этот оператор позволяет найти и вызвать метод по имени в указанном классе или в объекте. Этот оператор позволяет вызывать таким способом не только static методы от имени класса, но и объектные методы от имени объектов. Рассмотрим процесс вычисления суммарной стоимости рыб в коллекции с учетом этого оператора:

```
double cost1 = fishes.stream()
    .filter(f -> f.getPrice() > 100)
    .mapToDouble(Fish::getPrice)
    .sum();
System.out.println("cost1="+cost1+"  cost="+cost);
```

Вместо лямбда-выражения f -> f.getPrice(), использованного в первом случае в методе mapToDouble(), мы применяем ссылку на метод. Как видите, результаты вычисления такие же.

Вывод этого кода будет таким:

```
cost1=693.0  cost=693.0
```

Мы несколько раз перед этим использовали лямбда-выражение для вызова метода println():

```
.forEach(s->System.out.println(s));
```

Эти вызовы также можно заменить ссылкой на метод:

```
.forEach(System.out::println);
```

Надо еще отметить, что можно создавать потоки из набора значений не ссылочных, а примитивных типов int, double и других:

```
IntStream.of(1, 4, 11, 7, 32, 4, 79).forEach((c)->
          System.out.println(c));
```

Вы получили представление о том, что представляют собой лямбда-выражения и как их использовать. Возможно, продемонстрированные примеры кажутся вам непривычными, но вы должны почувствовать, что это не просто новый синтаксис. Лямбда-выражения – это новое качество языка Java.