**УРОК 2 – JAVA COLLECTIONS FRAMEWORK. ARRAYLIST**

На этом уроке мы начнем разбираться с Java Collections Framework и разберемся конкретно с классом ArrayList().

Для начала разберемся что такое JCF. JCF – это набор связанных классов и интерфейсов, реализующих commonly reusable collection структур данных. Commonly reusable collection переводится как часто использующиеся коллекции. То есть у нас существуют в проекте объекты и мы их хотим как-то хранить. То место, где мы храним объекты называется коллекцией. JCF – это просто набор классов по хранению объектов.

Начнем разбор фреймворка с разбора класса ArrayList. Класс ArrayList реализует это класс, который реализует функционал динамического массива, т.е. массива, который расширяется при добавлении новых объектов.

Давайте посмотрим, где находится класс ArrayList в иерархии JCF.



У нас есть общий родительский интерфейс, который называется iterable (перечисляемое с англ.). Т.е. во всех коллекциях у нас находятся объекты и все эти объекты мы можем перечислить. Все эти коллекции – это нечто перечисляемое.

Дальше от интерфейса наследуется интерфейс Collection, и от него наследуется три интерфейса: List, Queue и Set. Два последних мы разберем позже. Пока что разберемся в первом (List). Интерфейс List реализуют классы ArrayList, LinkedList и Vector, а класс Stack наследуется от класса Vector.

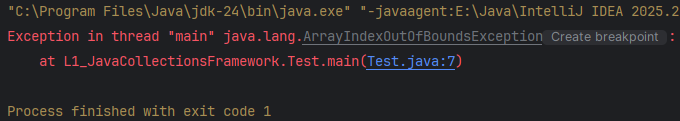
Пока не будем рассматривать нижние три класса. Начнем с ArrayList. Этот класс реализует интерфейс ArrayList.

Теперь, когда мы понимаем место ArrayList в иерархии Java Collections Framework, применим его на практике.

Создадим класс Test и в нем сначала создадим массив. Посмотрим, в чем мотивация была для создания класса ArrayList.

Размер массива сделаем 3 элемента, но попробуем поместить в него 4 элемента

public class Test {  
 public static void main(String[] args) {  
 int[] x = new int[3];  
 for (int i=0; i<4; i++){  
 x[i] = i;  
 }  
 }  
}



Мы получаем такую ошибку. Именно по этой причине и нужен ArrayList.

Все классы в JCF параметризованы, то есть при создании объектов этих классов, нам необходимо указать параметр, то есть тот тип объекта, который будет храниться в этой коллекции. Будем хранить там Integer.

ArrayList<Integer> arrayList = new ArrayList<Integer>();

После Java 7 мы можем не писать тип хранимых объектов в месте создания нового объекта. Достаточно указать его при декларации переменной:

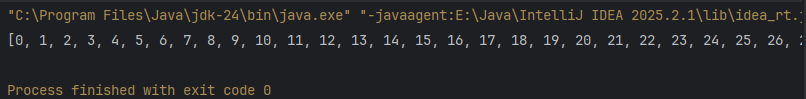
ArrayList<Integer> arrayList = new ArrayList<>();

ВАЖНО! В параметре объекта ArrayList можно указывать только ссылочные типы. Примитивы – нельзя! Это еще одна причина использования класса Integer.

Добавим объект в ArrayList. Делается это с помощью метода add(). У нашего объекта arrayList мы вызываем метод .add() и в параметрах метода указываем объект который хотим добавить . Добавим 100 элементов с помощью цикла for.

public static void main(String[] args) {  
 ArrayList<Integer> arrayList = new ArrayList<>();  
 for(int i = 0; i<100; i++){  
 arrayList.add(i);  
 }  
}

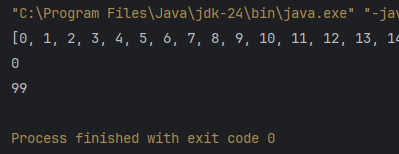
Теперь выведем arrayList на экран. В этом классе переопределен метод toString(), поэтому мы без проблем увидим наш массив:



Следующий метод, который мы можем получить – метод get.

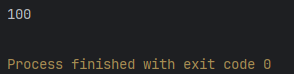
Работает это также как в массиве, только здесь с использованием метода. Мы с помощью этого метода можем получить элемент по индексу.

System.*out*.println(arrayList.get(0));  
System.*out*.println(arrayList.get(99));



Следующий метод – size() – аналог array.length

System.*out*.println(arrayList.size());



Как мы заметили, мы не указывали количество элементов при создании объекта ArrayList. Он подстраивается сам.



Единственное ограничение этого массива – это память компьютера, на котором выполняется программа.

Способы, как мы можем проходится по элементам ArrayList.

1. Первый способ – самый очевидный – с помощью цикла for

for(int i = 0; i<arrayList.size(); i++){  
 System.*out*.println(arrayList.get(i));  
}

1. Второй способ – for each

for(Integer i : arrayList){  
 System.*out*.println(i);  
}

ВАЖНО! Не рекомендуется делать так:

for(int i : arrayList){  
 System.*out*.println(i);  
}

По сути так тоже работает, но неявно происходит распаковка и на самом деле это выглядит так:

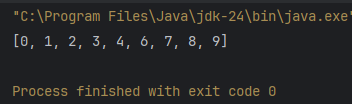
for (Integer temp : arrayList) {  
int i = temp.intValue(); // автораспаковка  
 System.out.println(i);  
}

Все работает, пока в arrayList не попадется null. В таком случае выпадет ошибка NullPointerException.

1. Третий способ пройтись по всем элементам массива – это пройтись по лямбда функциям. Но об этом будет рассказано позже, в одном из будующих уроков.

Еще один метод, который есть у всех объектов ArrayList это remove(). Для упрощения сократим количество элементов массива до 10. И удалим 5-ый элемент массива.

public static void main(String[] args) {  
 ArrayList<Integer> arrayList = new ArrayList<>();  
 for(int i = 0; i<10; i++){  
 arrayList.add(i);  
 }  
 arrayList.remove(5);  
 System.*out*.println(arrayList);  
}



Единственное, что стоит отметить по поводу метода ArrayList, это то, что он очень не эффективен, если происходит удаление какого-то элемента посередине или ближе к началу листа. Почему так происходит подробно будет в след. Уроках. Но если в кратце, то ArrayList – не какой-то магический класс. Он реализован средствами языка Java. И устроен он так, что внутри находится обычный массив на ограниченное количество элементов. И когда массив уже полный и мы создаем новый элемент, то создается новый массив, размером в два раза больше, и все элементы из старого маленького массива переписываются в новый. Таким образом добивается эффект динамичности.

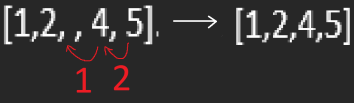
В случае же удаления элемента из ArrayLista, мы удаляем элемент из внутреннего массива. Пример:

У нас есть массива из 5 элементов: [1,2,3,4,5]

И мы хотим удалить последний элемент. Мы вызываем remove() и он просто уменьшает размер массива на 1. Получается [1,2,3,4]

Но что если мы хотим удлаить элемент под индексом 2 например. Remove() берет и удаляет этот элемент из массива – [1,2, , 4, 5]. Но нельяз оставлять массив в таком виде: с пустым элементом посередине.

Все элементы после удлаенного элемента переносятся на один шаг влево.

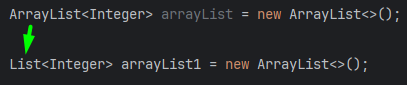


В больших массивах это занимает много времени. И чем ближе к началу удалаяемый элемент, тем больше операций по переносу элементов массива.

Для решения этой проблемы мы будем использовать LinkedList, который разберем в следующем уроке.

Пока нужно просто понять, что если в листе планируется много удалений, тем более в начале или середине, то лучше отказаться от исползования ArrayList в пользу LinkedList.

При использовании классов из Java Collections Framework, существует конвенция. И заключается она в том, что когда мы используем класс из Java коллекций, когда мы создаем объект этого класса, ссылку (переменную) на этот объект мы стараемся сделать типа интерфейса (в нашем случае List). Потому что класс ArrayList релаизует интерфейс List.



Это будет работать ровно также, как и работало. Зачем это нужно?

Переименуем нашу переменную и назовем ее просто list

Предположим, что в первой части программы нас устраивает реализация ArrayList, а во второй нам нужно много всего удалять.

В этом случае мы можем сослать нашу переменную на объект LinkedList. Только нужно не забыть в новый объект класса LinkedList перенести элементы из объекта класса ArrayList.

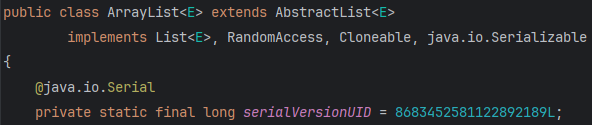
Это не сказать, что очень полезно, просто эта конвенция хороша, что в данном случае мы не думаем о реализации (типе переменной) нашего листа, и если нам это понадобится, то мы можем прямо в ходе программы поменять реализацию нашего листа.

**УРОК 3 – ARRAYLIST ПОД КАПОТОМ**

На этом уроке мы заглянем в реализацию arrayList и разберемся как он работает, как он с помощью стандартных средств джавы выполняет функцию динамического массива, т.е. расширяется при добавлении новых элементов.

Перейдем в реализацию класса ArrayList. Этот класс находится в пакете java.util.

Мы видим, что она параметризованный, он наследуется от AbstractList<E> и реализуется List<E>, RandomAccess, Cloneable, lava.io.Serializable. Здесь присутствует поле serialVersionUID.



У нас есть константная переменная, указывающая на стандартный размер массива:



Т.е., когда мы создаем объект класса ArrayList, у нас создается массив размером в 10 символов.

Спускаемся ниже и видим создание массива элементов класса Object elementData. Тип у него transient, как мы помним это значит, что этот массив не сериализуется, т.е. не добавляется в файл.

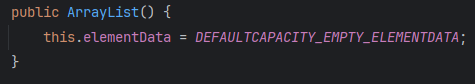


Есть также внутреннее поле size, которое хранит текущий размер нашего ArrayList.



И мы видим несколько конструкторов.

Вот конструктор, который мы использовали (без аргументов)



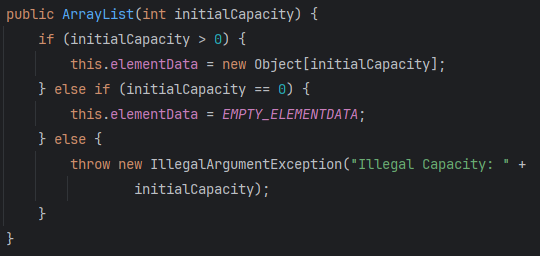
Нашему массиву присваивается по умолчанию пустой массив класса Object:



Мы также можем для нашего ArrayList сразу указать размер, чтобы сократить автоматические расширения массива и оптимизировать программу (если мы знаем что в массиве точно будет какое-то количество элементов):

List<Integer> list = new ArrayList<>(100);

Тогда вызывается конструктор с параметром int:



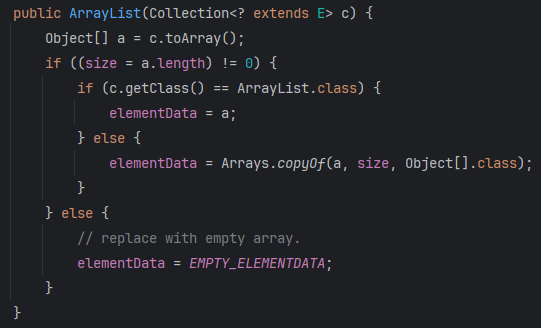
Если больше нуля, то создаем новый массив Object размером в переданный аргумент.

Если 0 – то пустой массив



Если аргумент отрицательный, то выбрасывается исключение.

И есть у нас еще один конструктор, который принимает на вход другую коллекцию. Этот конструктор используется тогда, когда есть у нас какой-то List, какая-то коллекция. ? означает, что мы используем Wildcard, таким образом мы можем передать List любых объектов.:

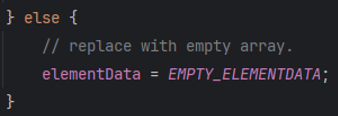


Сначала мы создаем переменную a типа Object[]. Это временный массив, в который мы занесем данные из ArrayList. Далее мы вызываем метод toArray() у переданной коллекции и присваиваем результат переменной a:



Далее в блоке if мы присваиваем переменной size длину массива Object[].

Если она равна 0, то массиву elementData присваивается массив EMPTY\_ELEMENTDATA, т.е. пустой массив.



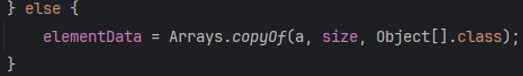
Если она не равна 0, то:



То мы переходим к следующему if блоку

Если переданная коллекция c является классом ArrayList (сравнивается класс c и ArrayLisy), то массиву elementData мы присвоим наш временный массива a.



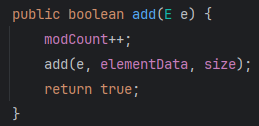
Иначе в наш массив elementData копируется коллекция любого другого типа. 

Подробно подкапотное метода copyOf разбирать не буду, там много пока не понятного.

Теперь пройдемся по методам ArrayList, которые мы использовали в предыдущем уроке.

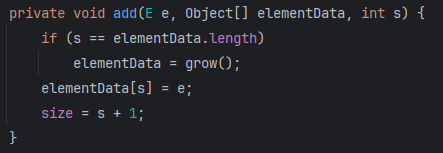
Add()

С начала найдем метод add.



В этот метод мы в качестве аргумента передаем тот элемент, который мы хотим добавить в наш ArrayList,

Из метода add мы передаем другой метод add с параметрами переданного элемента, внутреннего массива и размера объекта arrayLista.



Здесь если размер arrayList равен длине внутреннего массива, то внутреннему массиву присваивается увеличенное значение самого себя. Т.е. в методе grow происходит создание массива большего размера, а затем переноса туда элементов массива elementData.

Далее элементу[s] (например если длина arrayLista 25, то элементу с индексом 25 (это 26 элемент, т.к. подсчет с 0) внутреннего массива) присваивается переданный элемент и длина arrayLista увеличивается на 1.

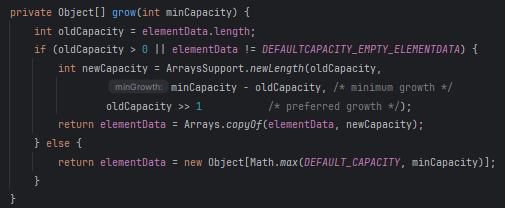
Как работает Grow() внутри?

Сначала как мы видим вызывается grow() без параметров. Внутри он выглядит так:



То есть внутри вызывается метод grow(), но уже с переданным аргументом size + 1(к текущему размеру ArrayList добавляется 1).

Метод grow(int minCapacity) внутри выглядит так:



1. Декларируем переменную oldCapacity (старый размер), в которую записываем текущую длину внутреннего массива
2. Если oldCapacity (старый размер) больше 0 или внутренний массив не является пустым по умолчанию (т.е. при создании не была указана размерность ArrayLista), то создается новый размер массива newCapacity. Он обращается к статическому методу newLength, который если отбросить все прочее возвращает нам значение oldCapacity \*1.5 (с отбрасывание дробной части). Почему так? Потому что как мы видим у нас фигурирует выражение oldCapacity >> 1. Знак >> означает побитовый сдвиг вправо. Например, oldCapacity = 25. 25 в двоичном коде это 11001. Мы сдвигаем все цифры на одну вправу и полчаем 01100 (1 пропадает, это и есть дробная часть которая отбрасывается). 1100 это 12.

Тогда наш newCapacity = 25+12=37.

Для примера рассмотрим еще одно число, к примеру, oldCapacity = 54. 54 в двоичном коде это 110110. Сдвигаем на один бит вправо -> 011011 -> это 27.

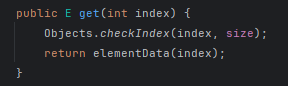
(Про если отбросить все прочее я говорил об очень больших числах, которые почти упираются в размерность типа int.)

1. Иначе (когда ArrayList создан без размера, т.е. он является пустым по умолчанию), этот вариант выполняется первым, если создан ArrayList без размера.

elementData становится массивом из 10 элементов (DEFAULT\_CAPACITY), если запрашиваемая емкость мест в массиве < 10. Или если больше 10, то создаем столько, сколько нужно (Больше может понадобится, когда все места запрашиваются разом. Через add() такое не сделать, но есть другие методы).

get()

Найдем метод add() в классе ArrayList.



Здесь на вход в параметры подается номер элемента (индекс).

Затем он проверяется в методе checkIndex(). Если попадает в размер, то все ОК. Возвращаем элемент массива по индексу.

Если укажем индекс выходящий за пределы массива – получим исключение.

size()

Найдем метод size().



Здесь все предельно просто. Возврат значения size. Это получается геттер.

Есть одна интересная деталь. В файле ArrayList.java есть два метода size и у них одинаковые сигнатуры. Как такое сожет быть? Второй size находится во вложенном классе. По этому все ОК.

remove()

Найдем remove, который принимает в параметры переменную типа int.

public E remove(int index) {  
 Objects.*checkIndex*(index, size);  
 final Object[] es = elementData;  
  
 @SuppressWarnings("unchecked") E oldValue = (E) es[index];  
 fastRemove(es, index);  
  
 return oldValue;  
}

С помощью метода checkIndex он проверяет есть ли такой элемент в массиве.

es – локальная ссылка на тот же самый массив, что и elementData.

То есть es и elementData указывают на один и тот же объект в памяти. Если мы поменяем es[i], то фактически изменится и elementData[i]. Это нужно для упрощения читаемости кода, для ускорения работы кода (так как к elementData идет обращение через this, а es – локальная переменная. Сейчас JIT умеет все оптимизировать, но н старых версиях JVM это давало небольшой прирост скорости. Поэтому в легаси коде такое может встречаться). Плюс такая переменная final, ее нельзя случайно переопределить позже, и понятно, что работа идет с этим конкретным массивом.

В методе fastRemove по переданным параметрам массива и индекса происходит удаление элемента массива (присваивание null) и сдвигание его элементов влево.

Почему метод возвращает удаленный элемент, а не массив без этого элемента?

Потому, что, если бы мы вернули измененный внутренний массив, мы бы нарушили инкапсуляцию. К внутреннему массиву ArrayLista можно было бы добраться вручную.

А так мы возвращаем удаленный элемент, что в принципе может быть полезно.

Ничего сложного в ArrayList нет, эта конструкция могла бы быть написана и самостоятельно. Но так как в java она достаточно часто используется, она была вынесена в коллекцию.

ВАЖНО! В процессе проверки работы внутренних классов я использую отладчик. Я ставлю точки останова в нужных местах, чтобы понять природу работы классов. НО когда я ставлю точку останова внутри любого класса java, отладчик не может сразу прикрепить точку останова – ведь кода этого класса еще нет в памяти. Поэтому точка останова срабатывает в момент подгрузки этого класса JVM, даже если код из него не вызывается.

JVM подгружает много классов уже при старте, поэтому отладчик мгновенно просыпается. Чтобы такого не было, нужно создать точку, до которой не нужно активировать отладчик. А затем во всех методах, что нас интересуют

указать эту точку, как точку, после которой можно будить отладчик.

Например: Нас интересуют конструкторы в классе ArrayList. По нашей логике к этим конструкторам мы должны перейти после строчки

List<Integer> list = new ArrayList<>(5);

В методе main класса Test.

Мы ставим точки останова на конструкторах в ArrayList:



на строчке в методе main:

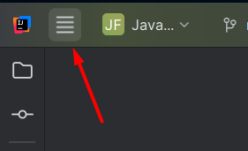


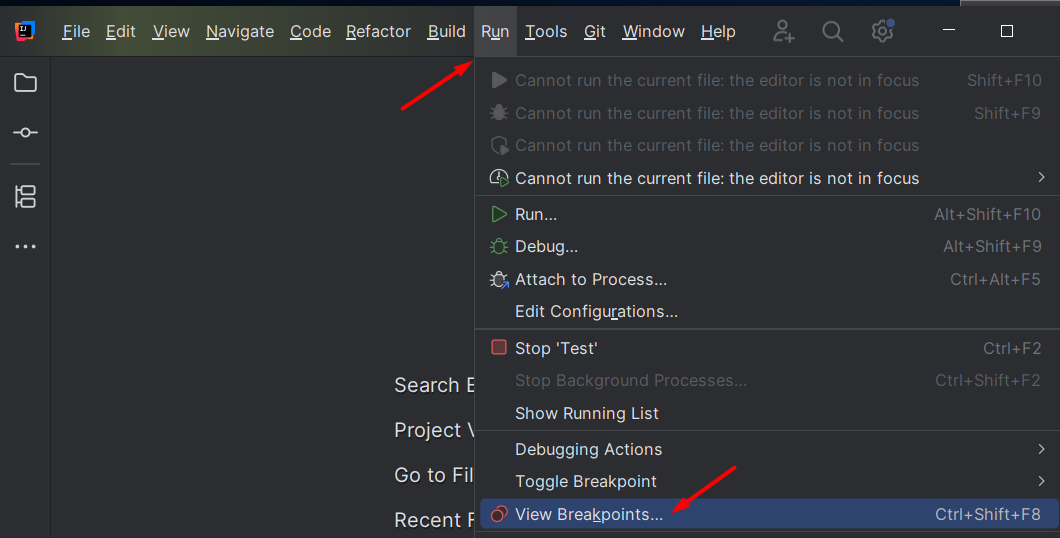
но когда запускам отладчик, то он останавливается на точках с конструктором еще не проходя метод main нашего класса Test.

И при переходе к следующим шагам отладки мы скачем туда обратно, от одного метода у другому (на самом деле в этот момент происходит подгрузка класса, создаются какие-то объекты, в общем происходит то, в создании чего мы участия не принимали).

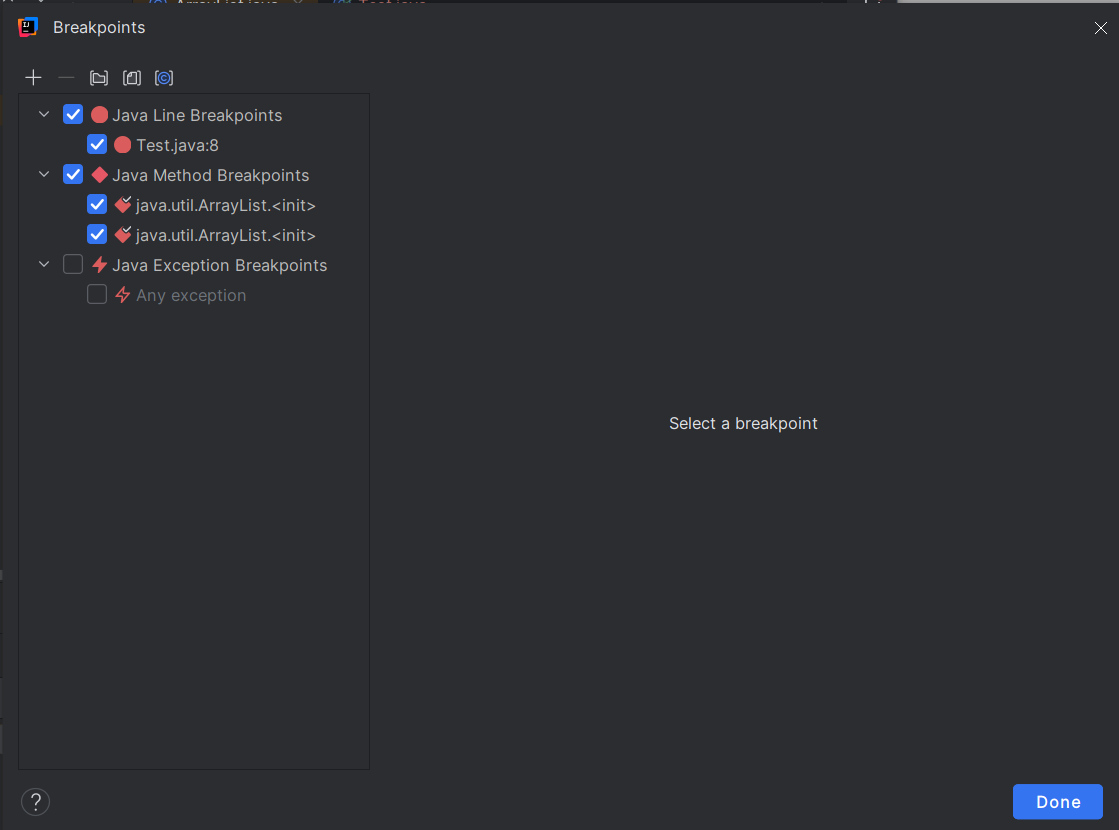
Через определенное количество скачков мы наконец добираемся до нашего метода main, это значит, что все прогружено и теперь начинается выполнение того, что писали мы.

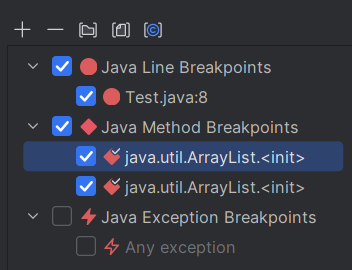
Чтобы у нас не было проблем с преждевременным вызовом отладчика, делаем следующее: ctrl+shift+f8 или Menu -> Run -> View Breakpoints.



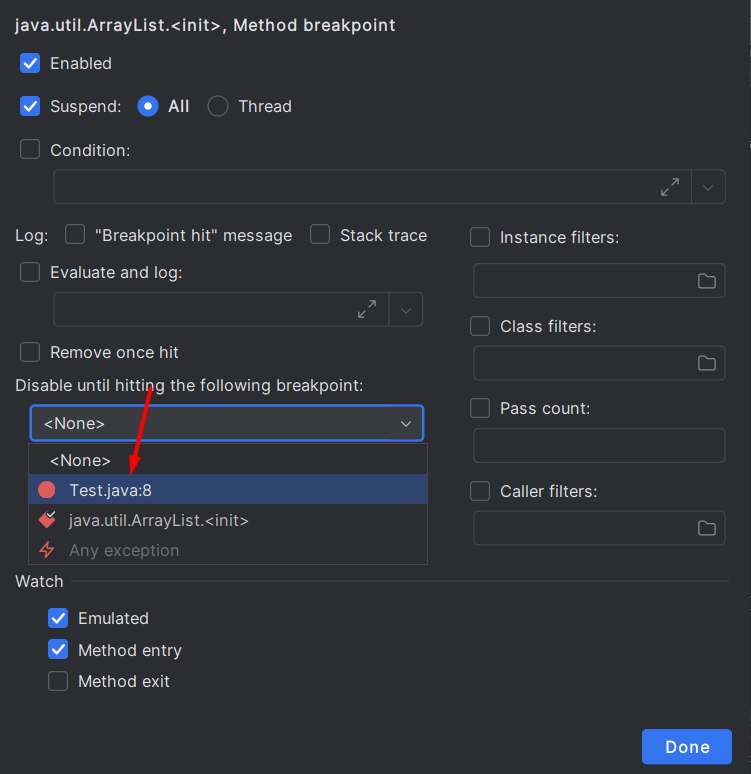


Открывается такая страница:

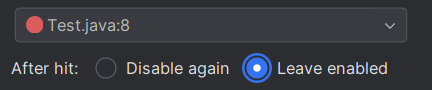
  
Теперь выбираем точку останова которую мы хотим до определенного усыпить при отладке



И в поле Disable until hitting the following breakpoint выбираем ту точку останова, до которой спящая точка не проснется.



И



Готово!

Либо можно просто расставлять точки, когда программа уже выполняется.

**УРОК 4 – LINKEDLIST – СВЯЗНЫЙ СПИСОК**

На этом уроке мы начнем изучать LinkedList в Java коллекциях. Рассмотрим использование этого класса и сравним его с ArrayList, а в следующем уроке разберем как это класс работает под капотом.

Создадим новый объект класса LinkedList.

Так как LinkedList и ArrayList реализуют интерфейс лист, у них одинаковые методы, которые описаны в интерфейсе List.

public class Test {  
 public static void main(String[] args) {  
 List<Integer> linkedList = new LinkedList<>();  
  
 linkedList.add(1);  
 linkedList.get(0);  
 linkedList.size();  
 linkedList.remove(0);  
 }  
}

Это очень хорошо что эти два разных класса имеют одну и ту же структуру. Т.е. между двумя этим классами можно легко переходить и программа не поменяется.

Уберем методы и создадим новый объект класса ArrayList.

Теперь создадим метод, который будет замерять время работы разных операций на LinkedList и ArrayList. В качестве аргументов будем передавать лист интеджеров. Еще одна хорошая сторона того, что оба эти класса реализуют интерфейс List, это то, что можно создавать для них универсальные методы, которые смогут принимать объект обоих этих классов.

public static void measureTime(List<Integer> list){  
   
}

Создадим переменную long start, которая будет возвращать текущее время в милисекундах.

long start = System.*currentTimeMillis*();

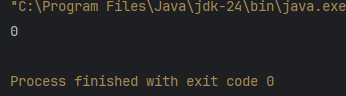
Получается, что у нас будет время в начале – до того как мы делаем какие-то операции, и будет время после проделанной операции

long end = System.*currentTimeMillis*();

И в конце между этими методами мы будем выводить время в милисекундах, которое прошло между инициализацией этих переменных.

public static void measureTime(List<Integer> list){  
 long start = System.*currentTimeMillis*();  
   
 long end = System.*currentTimeMillis*();  
   
 System.*out*.println(end - start);  
}

если мы сейчас вызовем и запустим этот метод, то мы получим 0. Так как между этими опреациями ничего не происходит и код выполняется практически мгновенно.

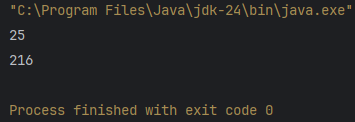


Теперь будем проверять скорость работы. В наш лист поместим миллион чисел с помощью цикла for и метода add().

public static void measureTime(List<Integer> list){  
 long start = System.*currentTimeMillis*();  
 for (int i=0; i<1000000; i++){  
 list.add(i);  
 }  
 long end = System.*currentTimeMillis*();  
 System.*out*.println(end - start);  
}

Теперь в методе main вызовем этот метод два раза. В первый раз передадим в качестве аргумента linkedList, а во второй раз – arrayList.

public static void main(String[] args) {  
 List<Integer> linkedList = new LinkedList<>();  
 List<Integer> arrayList = new ArrayList<>();  
  
 *measureTime*(arrayList);  
 *measureTime*(linkedList);  
}

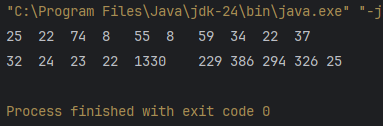


К сожалению результаты при тестировании метода для каждого листа по одному разы недостоверны. Если мы поменяем местами linkedList и arrayList, то результат будет схож, хотя в этот раз должно быть наоборот.

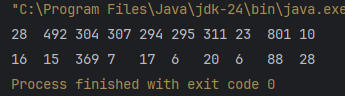


Замечена закономерность, что первая итерация метода с использованием наши классов всегда быстрее. Для того чтобы получить достоверную информацию, протестируем вызов каждого метода по 10 раз.

Сначала разместим arrayList, потом linkedLIst.



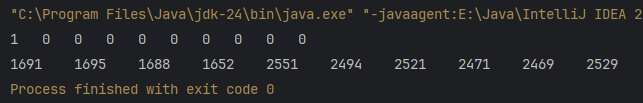
Поменяем их местами



Теперь заметно что arrayList справляется с **add()** быстрее. В среднем за эти две теста время выполнения у arrayList = 46 мс. У linkedList же 278 мс, т.е. в 6 раз быстрее. (При других количествах данный разница может отличаться, но факт, что arrayList для add использовать целесообразнее)

Теперь повторим эксперимент для метода **get().** Гет занимает больше времени, поэтому уменьшим количество элементов до 50000. В этот раз и дальше обойдемся сканированием только в одну сторону, этого будет достаточно, так как мы уже убедились в адекватности значений на методе add().

Сначала с arrayList, потом linkedList.



Разница колоссальна. Подсчитывать средние значения смысла нет.

Теперь вернемся к методу **add()** и попробуем добавлять данные в лист другим способом. Кол-во эл-тов - 50000

До этого мы добавляли элементы **в конец списка**.

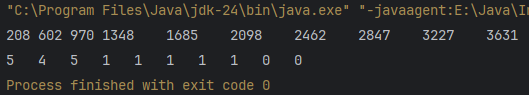
Теперь будем добавлять элементы в начало, сдвигая добавленные к концу.

У метода add() есть перегрузка, при которой он принимает два параметра: первый – это индекс, второй какой элемент мы помещаем.

list.add(0, i);

Также этот метод мы можем заменить на:

list.addFirst(i);



Как мы видим в этот раз в методе add() безоговорочно побеждает linkedList.

Почему так происходит? Потому что ArrayList и LinkedList имеют абсолютно разную реализацию.

Сначала посмотрим как arrayList проводил операцию вставки в конец цикла.

У нас был внутренний массив

[1][2][3][4][5][]

И когда мы хотим добавить 6, мы просто добавляем 6 и увеличиваем внутреннее поле size.

[1][2][3][4][5][6]

Случаи, когда внутренний

массив увеличивается из-за нехватки места и нам приходится создавать новый массив большего размера и копировать в него все элементы прошлого массива не очень частые. Поэтому в большинстве случаев добавление элементов в массив происходит быстро.

Что же мы делаем во втором случае, когда помещаем новые элементы в начало списка?

[1][2][3][4][5][]

Мы все элементы сдвигаем на один вправо, а затем добавляем элемент 6

[1][2][3][4][][5]

[1][2][3][][4][5]

[1][2][][3][4][5]

[1][][2][3][4][5]

[][1][2][3][4][5]

[6][1][2][3][4][5]

Это занимает очень много времени по сравнению с прошлым вариантом.

В случае linkedLista мы получаем очень хороший результат в случае добавления в начало списка. Потому, что linkedList устроен по-другому. Подробнее об этом в следующем уроке, но если в кратце, то в linkedLIst у нас хранится цепочка объектов. У каждого объекта есть значение и ссылка на следующий объект.

List<Integer> linkedList = new LinkedList<>();  
// head -> [1] -> [2] -> [3] -> [4] -> [5]  
List<Integer> arrayList = new ArrayList<>();  
// [1][2][3][4][5]

Рассмотрим разницу на примере метода **get()**:

В arrayList – массив, ссылки на элементы которого хранятся в одном месте. Если нам нужен 4-ый элемент массива, мы:

* идем по индексу 3.
* получаем ссылку на объект.
* идем к этому объекту.

В linkedList – нет массива. Каждый элемент linkedLista – это отдельная ячейка в памяти, содержащая значение и ссылку на следующую ячейку. И когда мы хотим получить 4 элемент массива, мы не модем сразу получит ссылку на него. Мы:

* сначала идем в 0-вой элемент листа.
* там получаем ссылку на 1-ый элемент.
* идем в 1-ый.
* там получаем ссылку на 2-ой.
* идем во второй.
* во втором получаем ссылку на 3-ий
* идем к объекту по индексу 3 (т.е. к 4ому объекту листа)

А теперь сделаем сравнение arrayList и linkedList для метода add() в начало списка. Рассмотрим наш пример представленный ранее с 5 элементами и добавлением 6-го в начало.

В arrayList :

* Находим последний элемент массива – 5, индекс 4
* Сдвигаем элемент 5 из ячейки с индексом 4 в 5
* Сдвигаем элемент 4 из ячейки с индексом 3 в 4
* Сдвигаем элемент 3 из ячейки с индексом 2 в 3
* Сдвигаем элемент 2 из ячейки с индексом 1 в 2
* Сдвигаем элемент 1 из ячейки с индексом 0 в 1
* Добавляем элемент 6 в ячейку 0.

В linkedList:

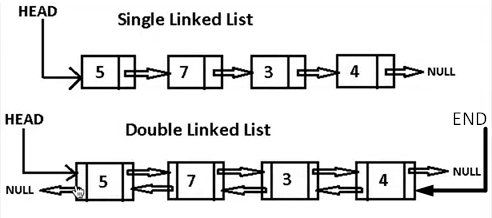
* Добавляем элемент 6, указываем в нем ссылку на следующий элемент (1).
* В головном элементе head, указываем ссылку на 6.

**УРОК 5 – LINKEDLIST – ПОД КАПОТОМ**

На этом уроке разберем, как работает linkedList внутри.

LinkedList – реализует структуру данных связного списка.

Есть два вида связных списков:



* Односвязный
* Двусвязный

В реализации LinkedList в Java реализуется двусвязный список.

Мы видим, что у двух этих типов списков есть отдельные узлы, т.е. это не массив.

В случае односвязного списка в узлах хранится ссылка только на следующий узел.

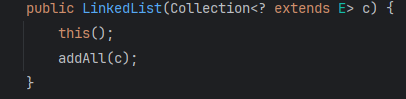
В случае двусвязного списка в узлах хранятся ссылки на следующий и на предыдущий узлы.

Двусвязный более эффективный. Если мы хотим в односвязном списке получить последний элемент, то нам нужно начать сначала и пройтись до самого конца. В двусвязном списке помимо HEAD есть также и END, т.е. начинать можно и с конца

Заглянем в реализацию LinkedList



Мы видим, что конструктор в нем пустой. Помимо него есть конструктор, который принимает другую коллекцию (как и ArrayList):

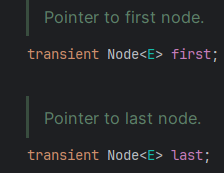


И в этом случае у нас строится LinkedList на основе той другой коллекции, которая было подана на вход в качестве аргументов.

Рассмотрим поля этого класса:



size – указывает на размер текущего листа

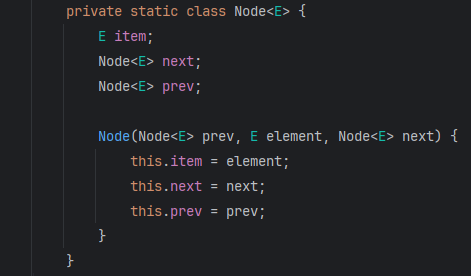


Есть узлы first и last, если перенести на картинку, то это HEAD И END.

Когда мы вызываем, например, метод get, мы можем пойти к элементу с двух сторон, смотря как будет быстрее.

Здесь у нас узлы параметризованы , мы можем слздавать узлы, которые хранят в себе разные значения. Их мы указывает в параметрах при создании обхекта класса LinkedList. Но мы создадим линклист, который может хранит только числа, это нужно чтобы была понятна суть и чтобы не загромождать код.

Теперь переместимся в реализацию класса Node.

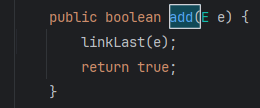


Мы видим, что это приватный статический вложенный класс.

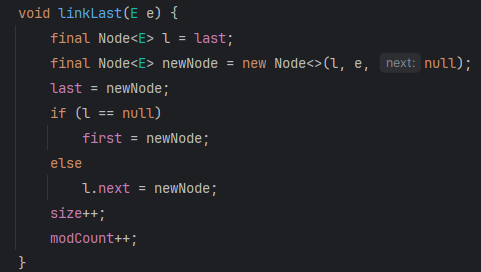
Внутри он хранит в себе сам элемент item типа E, следующий узел next и предыдущий узел prev.

Все эти значения поставляются в параметры конструктора при создании узла.

Теперь посмотрим как реализован тип add() в линклисте



В нем есть метод linkLast.



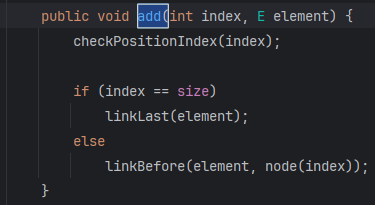
В этом методе, если не вдаваться в подробности создается новый узел (новый объект класса Node), которому в параметры передаются: узел last(то, что мы называли END), значение e, и null для узла next(так как это последний элемент.

На 3 строчке тела метод узлом last становится только что созданный узел.

Дальше если l == null (т.е. last, который был присвоен объекту l, до этого момента не существовал), то текущий узел становится first.

Иначе: поле next у объекта l (предыдущего последнего) меняется на newMode, т.е. на ссылку на наш новый только что созданный узел.

Как и в ArrayList, мы можем добавить элемент не в конец или начало, а по индексу.



В таком случае у нас есть if else конструкция, которая, если введен последний индекс в листе, отправит нас по уже знакомому методу linkLast. Если нет, то мы перейдем в метод linkBefore(опустим его рассмотрение, там ничего сложного), где также создается новый объект класса Node, в предыдущем узле меняется ссылка next (теперь она указывает на только что созданный узел), а в следующем меняется ссылка prev (теперь и она указывает на только что созданный узел).

Есть также метод addFirt. Действует аналогично с linkLast.



|  |  |
| --- | --- |
|  |  |

ВАЖНО! Индексы в linkedList работают не так как в ArrayList. В arraylist по индексу мы можем сразу перейти к элементу листа, а в linkedList по индексу мы знаем через какое количество элементов нам нужно пройти, чтобы добраться до нужного нам. То есть, например, в linkedList у нас 15 элементов. И мы хотим получить (get) значение 6-го элемента. Для этого класс LinkedList смотрит на узел HEAD, в котором содержится ссылка на 0 элемент, переходит к 0 элементу, там смотрит на ссылку на 1-ый элемент, переходит к первому элементу, смотрит на ссылку на 2-ой и так 6 раз (столько раз, какой индекс в get мы указали.

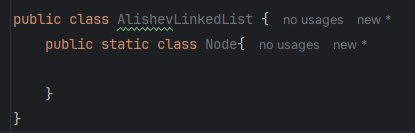
Когда классу LinkedList нужно понять с какой стороны ему двигаться ближе: с головы или хвоста, он берет длину этого листа и делит пополам (с помощью “>>” – битовый сдвиг вправо). Если получившееся число меньше, чем введенный нами индекс, то нужный нам элемент находится ближе к концу и добираться до него класс начнет с конца. А если получившееся число больше, то искомый элемент ближе к началу листа.

Теперь самостоятельно реализуем аналог класса LinkedList. Этот класс будет реализовывать односвязный список и он будет не параметризован. То есть на вход можно подавать только Integer – целые числа

Я создал свой LinkedList. Он находится в папке  и называется MyLinkedList. Класс, в котором находится метод main называется Test. В нем создается объект класса MyLinkedList, из него мы вызываем методы объекта, в нем мы изменяем этот объект. Здесь я его не рассматриваю, так как написал его сам. Разобраться в том что там написано можно самостоятельно без проблем.

Но помимо своего линкедЛиста, я напишу линкедЛист, который показывает у себя в уроке Алишев.

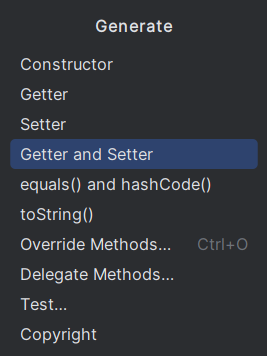
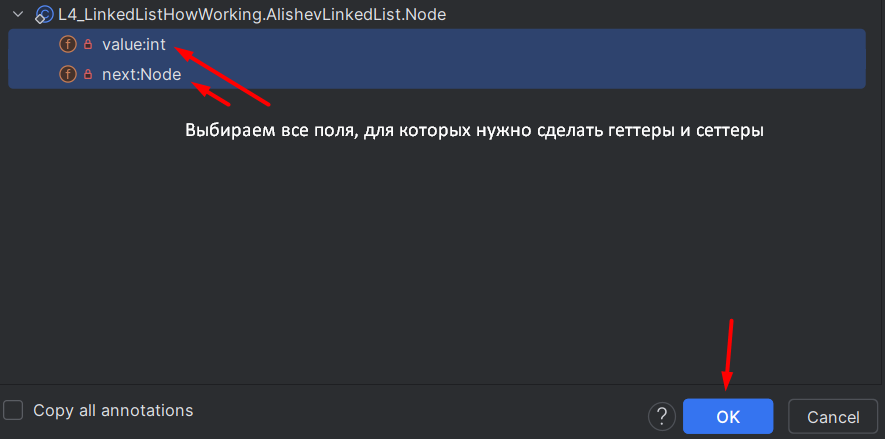
Сразу же внутри созданного класса создадим вложенный статический класс Node



Опишем класс Node. В нем будет храниться поле Node next, в этоми поле будет находиться ссылка на следующий объект в списке. А также будет поле int value, в котором будет храниться значение.

**Автоматическое создание getter, setter (геттеров и сеттеров).**

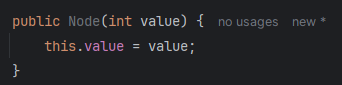
Создадим геттеры и сеттеры для каждого из этих полей. Для этого воспользуемся их автоматической генерацией (Alt + Insert)



**Автоматическое создание констурктора**

И создадим конструктор, также автоматически:



Конструктор нам нужен, так как при создании новых элементов листа мы будем передавать значение.

Конструктор в классе LinkedList создавать смысла нет, так как самому LinkedList мы ничего не передаем, мы только вызываем методы у его объектов.

Теперь добавим методы. Начнем с метода add. Этот метод ничего не будет возвращать, т.е. тип – void.

Для того, чтобы добавить элемент, нам нужна голова списка. Для этого создаем приватное поле Node head.

Если head == null (т.е. если это первый элемент в списке), то у объекта класса AlishevLinkedList, у которого мы вызвали метод add. сошлем поле head на новый объект Node.

public void add(int value){  
 if (head == null){  
 this.head = new Node(value);

Также создадим переменную size, которая будет хранить размер массива и будем добавлять к ней 1 каждый раз, когда вызывается метод add.

public void add(int value){  
 if (head == null){  
 this.head = new Node(value);  
 } else {  
  
 }  
 size++;  
}

Теперь сделаем логику работы при не первом элементе в списке. Мы должны дойти до последнего узла списка и должны нашему последнему узлу head в качестве следующего узла next указать новый узел.

Сначала дойдем до конца списка. Для этого мы должны создать временный узел и указать его на узел head.

} else {  
 Node temp = head;  
}

Теперь на наш головной узел указывает две ссылки.

Сейчас досчитаем до конца списка:

while (temp.getNext() != null){  
 temp = temp.getNext();  
 }

Для чего же нам temp, если можно также досчитать до конца списка, используя head? Ответ: head всегда указывает на первый элемент списка, а add добавляет новый узел в конец списка. (в моей версии реализации в качестве head выступает first, а в качестве temp - lastElement).

Когда мы уже добрались до последнего узла, мы можем создать следующий новый узел, которому передадим значение value, переданное в метод add.

temp.setNext(new Node(value));

Таким образом мы расширили наш список.

Весь метод add:

public void add(int value){  
 if (head == null){  
 this.head = new Node(value);  
 } else {  
 Node temp = head;  
 while (temp.getNext() != null){  
 temp = temp.getNext();  
 }  
 temp.setNext(new Node(value));  
 }  
 size++;  
}

Теперь реализуем метод toString, чтобы в удобном виде просматривать содержимое нашего списка.

Мы переопределим метод toString в классе Object, так как метод именно этого класса неявно вызывается, при выполнении метода println().

Чтобы вывести на экран содержимое нашего списка, мы сначала должны пройтись по нашему односвязному списку и куда-то записать то, что там у нас хранится.

Воспользуемся для этого массивом.

public String toString(){  
 int[] result = new int[size];  
}

Теперь создадим переменную index = 0 и переменную temp = head.

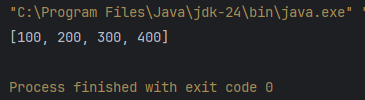
int index = 0;  
 Node temp = head;

Теперь пройдемся по всему нашему списку и выведем результат на экран:

while (temp != null){  
 result[index] = temp.getValue();  
 index++;  
 temp = temp.getNext();  
}  
return Arrays.*toString*(result);

Теперь проверим что у нас получилось. Пойдем в класс Test, создадим объект нашего AlishevLinkedList, и добавим в список несколько элементов, а затем выведем их на экран.

public class AlishevTest {  
 public static void main(String[] args) {  
 AlishevLinkedList myList = new AlishevLinkedList();  
 myList.add(100);  
 myList.add(200);  
 myList.add(300);  
 myList.add(400);  
 System.*out*.println(myList);  
 }  
}



Теперь реализуем метод get

Снова создадим временную переменную temp = head и переменную currentIndex = 0, которая будет считать до переданного в параметры индекса.

Node temp = head;  
int currentIndex = 0;

Теперь сделаем цикл: Пока temp != null (т.е. пока узел содержит значение, то бишь находится в списке): выполняем конструкцию if else.

Если currentIndex != index, то инкрементируем currentIndex и переменной temp присваиваем следующее значение.

Иначе (когда currentIndex становится равен index), мы возвращаем текущее значение узла.

while (temp != null){  
 if (currentIndex != index){  
 currentIndex++;  
 temp = temp.getNext();  
 } else{  
 return temp.getValue();  
 }  
}

Если переданный в параметры метода индекс слишком большой и итераций больше чем узлов в списке, то цикл while перестанет выполняться и вызовется исключение

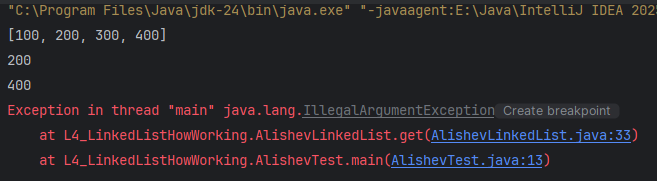
throw new IllegalArgumentException();

Весь метод выглядит так:

public int get(int index){  
 Node temp = head;  
 int currentIndex = 0;  
 while (temp != null){  
 if (currentIndex != index){  
 currentIndex++;  
 temp = temp.getNext();  
 } else{  
 return temp.getValue();  
 }  
 }  
 throw new IllegalArgumentException();  
}

Проверим:

public class AlishevTest {  
 public static void main(String[] args) {  
 AlishevLinkedList myList = new AlishevLinkedList();  
 myList.add(100);  
 myList.add(200);  
 myList.add(300);  
 myList.add(400);  
 System.*out*.println(myList);  
 System.*out*.println(myList.get(1));  
 System.*out*.println(myList.get(3));  
 System.*out*.println(myList.get(4));  
 }  
}



Теперь реализуем метод remove. В качестве аргумента ему будет передаваться индекс. Функционал похож с get. Нам нужно дойти до элемента и удалить его. Отличие заключается в том, что дойти нам нужно не до самого элемента, а до его предшественника. Возьмем наши 4 элемента:

100 -> 200 -> 300 -> 400.

Если нам нужно удалить узел 300, то если мы до него дойдем, то мы не сможем никаким образом удалить, ведь ссылка в нем находится на узел 400. А вот на элемент 300 ссылка находится в узле 200. Чтобы удалить элемент 300, нужно просто в узле 200 поменять ссылку на следующий элемент с 300 на 400.

На узел 300 больше ничего не будет ссылаться, и он просто удалится сборщиком мусора.

Скопируем код из метода get но с некоторыми изменениями:

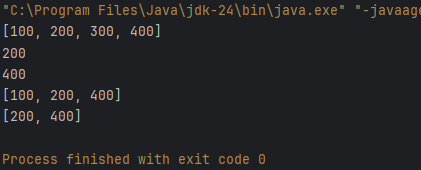
public void remove(int index) {  
 if (index == 0){  
 head = head.getNext();  
 size--;  
 return;  
 }  
 Node temp = head;  
 int currentIndex = 0;  
 while (temp != null){  
 if (currentIndex+1 != index){  
 currentIndex++;  
 temp = temp.getNext();  
 } else{  
 temp.setNext(temp.getNext().getNext());  
 size--;  
 return;  
 }  
 }  
}

Так как нам нужно добраться не до самого узла, а до узла перед ним, в условии if добавляем currentIndex != index-1. То есть мы доходим до элемента, который предшествует тому, который мы хотим удалить.

А в теле else мы меняем поле next на узел через один, таким образом на узел 300 больше ничего не ссылается. Уменьшаем размер массива с помощью size—и выходим из цикла с помощью return; после исключения узла из списка.

Еще у нас появилось условие if в самом начале, оно отвечает за удаление элемента списка под индексом 0. Если индекс = 0, то мы просто переназначаем голову на следующий элемент и все. Тогда все оставшееся тело метода можно не выполнять.

public class AlishevTest {  
 public static void main(String[] args) {  
 AlishevLinkedList myList = new AlishevLinkedList();  
 myList.add(100);  
 myList.add(200);  
 myList.add(300);  
 myList.add(400);  
 System.*out*.println(myList);  
 System.*out*.println(myList.get(1));  
 System.*out*.println(myList.get(3));  
 myList.remove(2);  
 System.*out*.println(myList);  
 myList.remove(0);  
 System.*out*.println(myList);  
 }  
}



LinkedList в java работает примерно также, только в java LinkedList двухсвязный, т.е. в каждом узле есть ссылка не только на следующий, но и на предыдущий элемент. И второе отличие в том, что связный список в java параметризован, т.е. принимать может не только целочисленные значения.

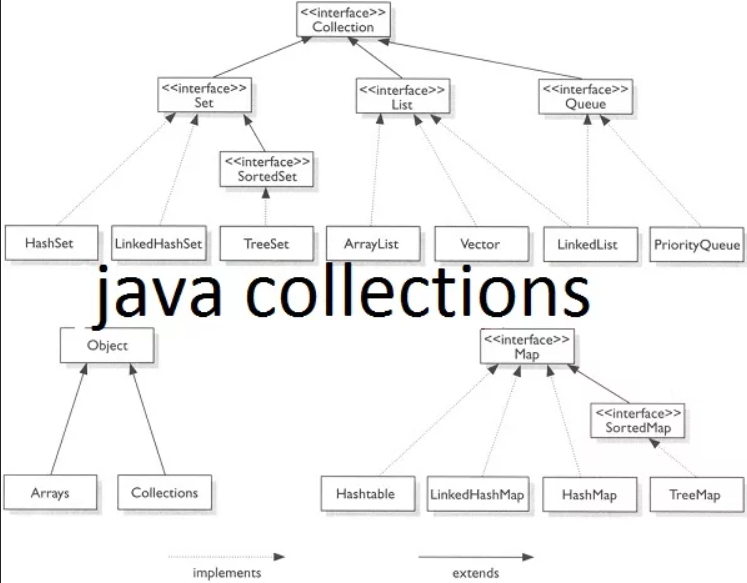
**УРОК 6 – HASHMAP - ВВЕДЕНИЕ**

На прошлых уроках мы разобрали принцип работы LinkedList и ArrayList интерфейса List.



Stack и Vector разберем попозже. А пока что перейдем к интерфейсу Map.

Эта картинка оказалась неполной, на ней нет интерфейса Map. На самом деле java collections framework немного побольше:



На этой картинке более полно представлены java collection framework. Но помимо классов реализующих интерфейс collection, есть еще класс, реализующий интерфейс Map. Именно с него мы и продолжим разбираться. Почему? Потому что по мнению Алишева классы реализующие Interface Map вторые по популярности по количеству использования после классов реализующих интерфейс List, после ArrayList и LinkedList.

Поэтому пока оставим классы, реализующие интерфейс collection и перейдем к Map.

Самый полезный и часто используемый класс, который реализует интерфейс Map это класс HashMap.

Map переводится с английского как 1. Карта, 2. Отображение.

Второй вариант более уместен в случае с этим интерфейсом. Отображение в математика – это случай, когда у нас есть два множества и каждому элементу из первого множества ставится в соответствие ровно один элемент из второго множества.

Пример: предположим у нас есть таблица. Слева в этой таблице у нас номера классов, а справа каждому классу сопоставляются ученики.

// слева класс - справа массив учеников  
// 1 -> [.. .. ..]  
// 2 -> [.. .. ..]  
// 3 -> [.. .. ..]

И структура данных HashMap (и в целом интерфейс Map) позволяет нам реализовать такую структуру данных.

Создадим объект класса HashMap

Как в случае с классами интерфейса List, слева пишем интерфейс, а справа создаем объект

Map<> hm = new HashMap();

При создании объекта класса HashMap, мы должны в треугольных скобках указать тип ключа и тип значения. Ключ в нашем примере – это номер класса, а значение – это массив с учениками. Т.е. все классы, реализующие интерфейс Map в java хранят пары: ключ – значение.

В нашем примере в качестве ключа мы будем использовать целые числа, а в качестве значения строки.

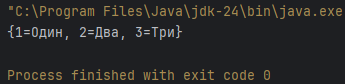
Map<Integer,String> hm = new HashMap<>();

Теперь положим в наш Map данные. Для этого используем метод **put()**. Этот метод принимает два элемента. Первый элемент – это ключ, а второй – это значение, которое мы хотим связать с этим ключом. Предположим, что в нашей таблице мы хранить целое число и перевод целого числа на русский язык. Положим 1 и перевод “Один”, 2 и “Два”, 3 и “Три”. ((В питоне конструкция map называется dictionary – словарь, у нас и получился словарь))

map.put(1, "Один");  
map.put(2, "Два");  
map.put(3, "Три");

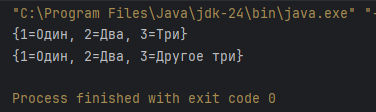
Выведем на экран

System.*out*.println(map);



Теперь проверим как себя ведет пара ключ значение, если мы в существующий ключ вставим новое значение.

Map<Integer,String> map = new HashMap<>();  
map.put(1, "Один");  
map.put(2, "Два");  
map.put(3, "Три");  
System.*out*.println(map);  
map.put(3, "Другое три");  
System.*out*.println(map);



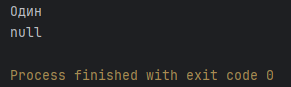
После вставки другого значение в существующий ключ, старое значение переписывается новым значением. Т.е. в map не может быть дубликатов ключей.

Теперь используем другой метод, который существует в map: **get()**. В этом методе мы можем получить значение по ключу

System.*out*.println(map.get(1));

Если пары с введенным ключом нет, то вернется null

System.*out*.println(map.get(6));



Теперь научимся проходиться по всем элементам map. Для этого воспользуемся циклом for each

for(Map.Entry<Integer, String> entry : map.entrySet()){  
 System.*out*.println(entry.getKey() + " : " + entry.getValue());  
}

Что здесь происходит? Дело в том, что каждая пара ключ – значение в java – это entry. Т.е. пара число-строка это entry. В map ключ значение не рассматриваются друг от друга отдельно, они рассматриваются как отдельная структура данных.

Сравним с обычным циклом for each, который был пройден ранее:

strings[0] = "один";  
strings[1] = "два";  
strings[2] = "три";  
for(String string:strings){  
 System.*out*.println(string);  
}

Для Array: Сначала у нас String - тип переменной, которой в процессе выполнения цикла присвоятся все значения массива.

Для Map: тип переменной - Map.Entry<Integer, String>. Entry – это интерфейс вложенный в интерфейс Map. Entry параметризован, то есть в качестве типов-аругментов ему передаются типы Integer и String

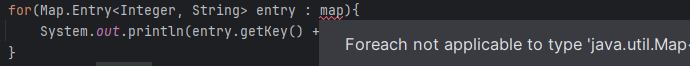
Для Array: string – название переменной, которой будут присваиваться все значения массива по порядку в цикле.

Для Map: entry – название переменной, которой будут присваиваться все значения хешмепа по порядку в цикле.

Для Array: strings – сам массив, который будет перебираться в цикле.

Для Map: map.entrySet()– это нестатический метод класса Map,

Если мы оставим только map, то компилятор выбьет нам ошибку:



Т.е. foreach мы не можем использовать с переменной типа Map, потому, что Map не реализует Iterable, значит возможности “обходить элементы” циклом for each у этого интерфейса нет.

Тогда почему работает map.entrySet()?

Потому, что entrySet() возвращает Set<Map.Entry<K,V>>, а Set

* является коллекцией
* реализует Iterable
* может участвовать в foreach

Именно поэтому цикл работает:

getKey и getValue – возвращают значение ключа и значения соответственно у текущего entry.

ВАЖНО! У класса хешмэп есть одна особенность, класс хешмэп не гарантирует порядка.

То есть мы можем по порядку ввести в него данные, а вывод придет совершенно в другом порядке. Поэтому если важно

Если номеров мало и расположены они по порядку, то они могут располагаться по порядку, но это случайность, а не функциональность. Их hashCode() равен самому числу, поэтому хэш-функция дает для них возрастающие бакеты, внутри одной таблицы ничего не перемешивается.

Если добавить ключи хаотично, то они наверняка сломают порядок.

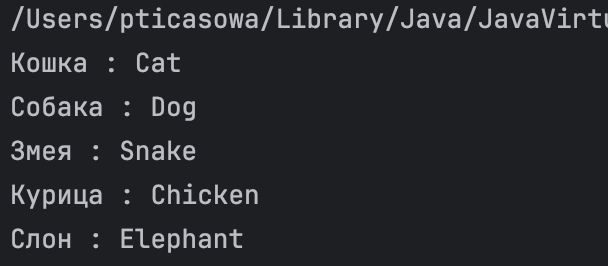
**УРОК 7 – LINKEDHASHMAP И TREEMAP**

Если в кратце, то LinkedHashMap сохраняет порядок добавления. А класс treeMap сортирует пары ключ-значение по ключу.

Здесь у нас код

public class Test {  
 public static void main(String[] args) {  
 Map<String, String> translations = new HashMap<>();  
  
 translations.put("Кошка", "Cat");  
 translations.put("Собака", "Dog");  
 translations.put("Слон", "Elephant");  
 translations.put("Змея", "Snake");  
 translations.put("Курица", "Chicken");  
  
 for (Map.Entry entry : translations.entrySet()){  
 System.*out*.println(entry.getKey() + " : " + entry.getValue());  
 }  
 }  
}

При запуске программы мы получаем:



Наши значния перепутаны. Они выводятся в другом порядке.

Теперь создадим 3 объекта разных классов: хешмэп, линкедлист и тримэп, а также метод, в котором будем тестировать эти 3 объекта.

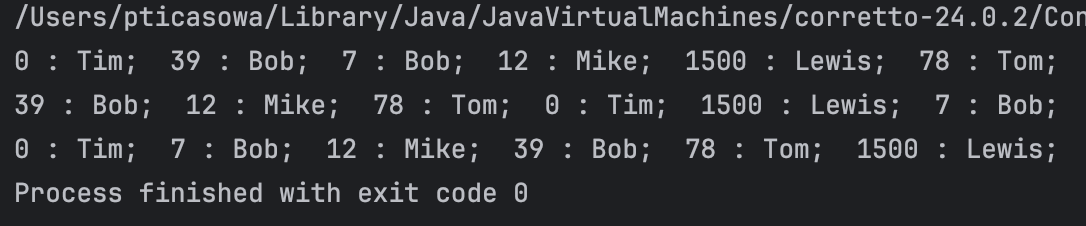
Map<Integer, String> hashMap = new HashMap<>(); //внутри не гарантируется порядка  
Map<Integer, String> linkedHashMap = new LinkedHashMap<>(); //гарантируется порядок добавления пар ключ-значение. Как добавили, так и вернутся  
Map<Integer, String> treeMap = new TreeMap<>(); // пары ключ-значение сортируются по ключу. Причем для ключа должен быть задан какой-то критерий сортировки (естественный порядок). Для целых чисел - понятно, для строк - сложнее, но в Java будет по алфавиту. Для своих объектов сортировку рассмотрим в следующих уроках

public void testMap(Map<Integer,String> map){  
}

И в этот метод мы поместим случайное значение. В качестве ключа мы будем помещать какое то случайное число, а в качестве значения будут какие нибудь имена людей.

public static void testMap(Map<Integer,String> map){  
 map.put(39, "Bob");  
 map.put(12, "Mike");  
 map.put(78, "Tom");  
 map.put(0, "Tim");  
 map.put(1500, "Lewis");  
 map.put(7, "Bob");  
 for(Map.Entry<Integer,String> entry : map.entrySet()){  
 System.*out*.print(entry.getKey() + " : " + entry.getValue() + "; ");  
 }  
}

Теперь из метода main вызовем этот метод от имени трех ранее созданных объектов:



Видим, что в первой строке у нас все перепуталось. Вторая строка не поменялась. А в третьей строке все элементы у нас стали по порядку по возрастанию ключа.

**УРОК 8 – МНОЖЕСТВА SET**

На прошлых уроках мы разобрали работу интерфейсов List и Map. Теперь разберем, что представляет из себя Set.

Set в переводе означает множества. Set - это коллекция, которая хранит в себе только уникальные элементы, в отличие от листа, в котором мы можем хранить сколько угодно дубликатов.

И это соотносится с теорией множеств в математике. В теории множеств нас интересует только набор элементов, как множество. Нас не интересует количество элементов. Нас интересует только есть ли элемент в этом множестве или нет.

Теперь создадим три объекта Set по полной аналогии с Map.

Set<String> hashSet = new HashSet<>();  
Set<String> linkedHashSet = new LinkedHashSet<>();  
Set<String> treeSet = new TreeSet<>();

List и Set во многом похожи. Они хранят одиночные элементы (без пары ключ-значение, как в случае с Map).

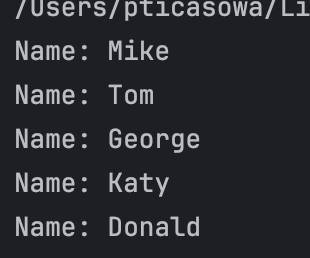
Только в Set в отличие от List нельзя хранить одинаковые элементы.

Теперь в наш hashset положим несколько элементов и посмотрим как HashSet работает.

hashSet.add("Mike");  
hashSet.add("Katy");  
hashSet.add("Tom");  
hashSet.add("George");  
hashSet.add("Donald");

Выведем их на экран воспользовавшись циклом foreach

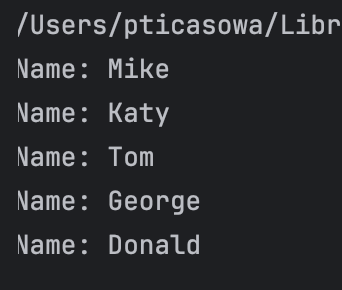
for(String name : hashSet){  
 System.*out*.println("Name: " + name);  
}



Поменяем hashset на linkedhashset.

Set<String> linkedHashSet = new LinkedHashSet<>();

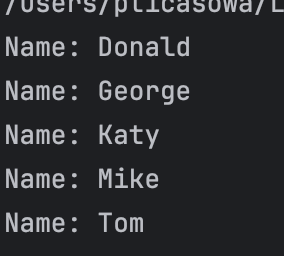
linkedHashSet.add("Mike");  
linkedHashSet.add("Katy");  
linkedHashSet.add("Tom");  
linkedHashSet.add("George");  
linkedHashSet.add("Donald");  
for(String name : linkedHashSet){  
 System.*out*.println("Name: " + name);  
}



И на treeset

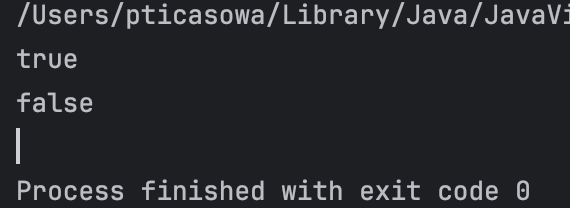
Set<String> treeSet = new TreeSet<>();

treeSet.add("Mike");  
treeSet.add("Katy");  
treeSet.add("Tom");  
treeSet.add("George");  
treeSet.add("Donald");  
for(String name : treeSet){  
 System.*out*.println("Name: " + name);  
}



Наиболее часто используемый метод в Set это метод contains. Он проверяет есть ли элемент в сете или нет. Он возвращает булевскую переменную.

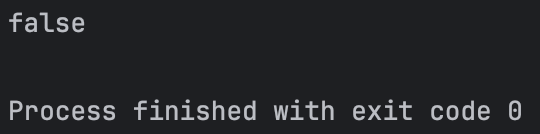
System.*out*.println(hashSet.contains("Tom"));  
System.*out*.println(hashSet.contains("Tim"));



Важно! В set метод contains работает очень быстро, происходит это потому, что используется хеширование. Об этом в одном из будующих уроков.

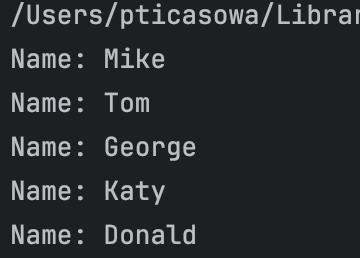
Есть также метод isEmpty, который возвращает true, если set пустой и false, если нет.

System.*out*.println(hashSet.isEmpty());



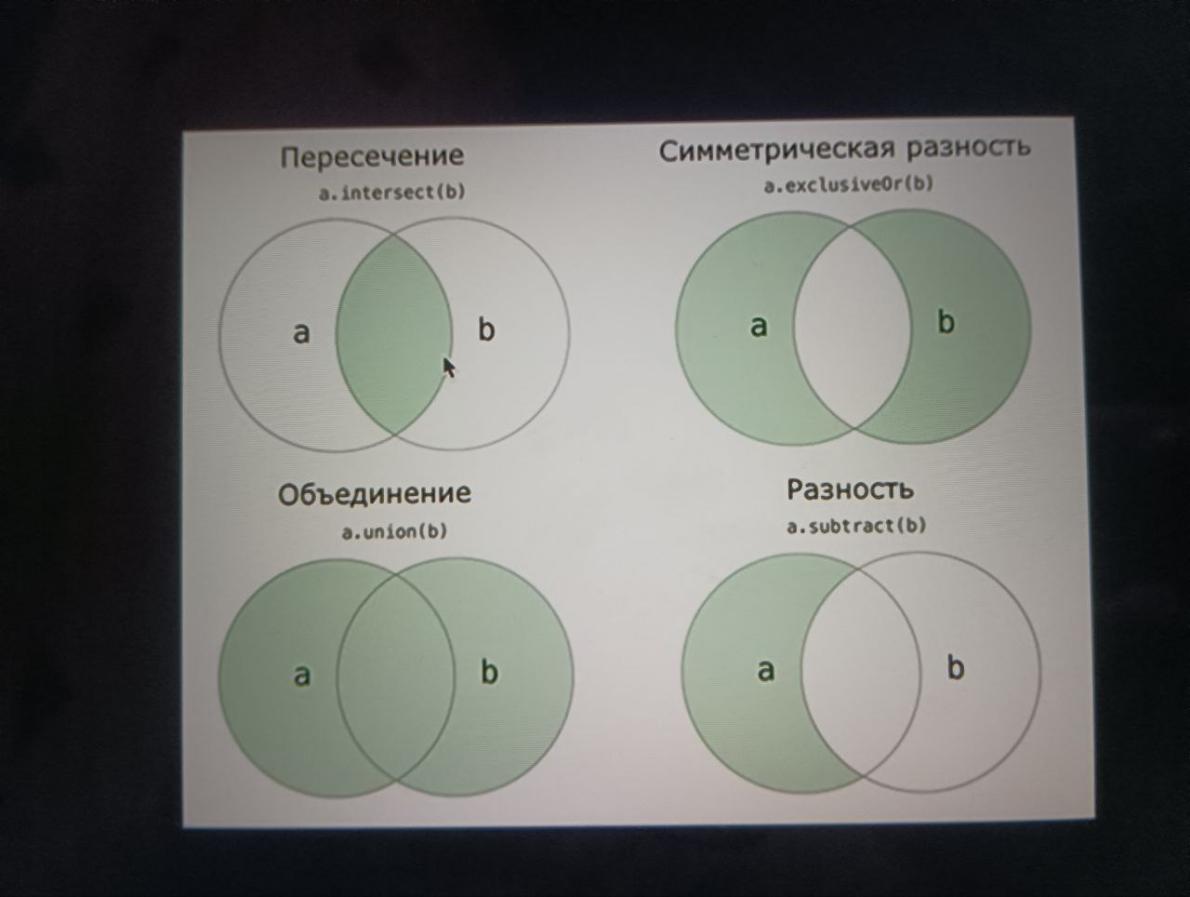
Если мы поместим в хешсет элементы, которые уже там есть, то таких элементов там больше не станет.

hashSet.add("Mike");  
hashSet.add("Katy");  
hashSet.add("Tom");  
hashSet.add("George");  
hashSet.add("Donald");  
hashSet.add("Tom");  
hashSet.add("Tom");  
for(String name : hashSet){  
 System.*out*.println("Name: " + name);  
}



Мы также можем выводить содержимое сета сразу через sout, потому что все классы, реализующие интерфейс set, реализуют метод

Еще одно применение set в java. В математике есть раздел теория множеств, который занимается и в этом разделе определяются операции, которые мы можем проводить над множествами.



Реализуем эти операции, начнем с **ОБЪЕДИНЕНИЯ**. Создадим 3 множства. 2 из них мы будем объединять и занесем объединение в третье множество.

Set<Integer> hashSet = new HashSet<>();  
Set<Integer> hashSet2 = new HashSet<>();

hashSet.add(0);  
hashSet.add(1);  
hashSet.add(2);  
hashSet.add(3);  
hashSet.add(4);  
  
hashSet2.add(3);  
hashSet2.add(4);  
hashSet2.add(5);  
hashSet2.add(6);  
hashSet2.add(7);

Set<Integer> union = new HashSet<>(hashSet);

В последней строчке мы перегружаем конструктор класса HashSet, передавая ему в качестве аргумента объект этого-же класса, и, как следствие, новый объект будет содержать все элементы переданного объекта.

union.addAll(hashSet2);

Метод addAll позволяет объединять сеты.

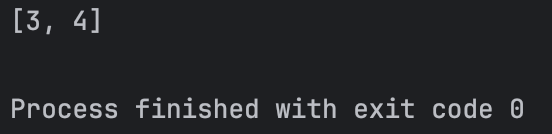


Теперь реализуем метод **ПЕРЕСЕЧЕНИЯ**

Точно также передадим метод hashset в конструктор класса HashSet.

Set<Integer> intersection = new HashSet<>(hashSet);  
intersection.retainAll(hashSet2);

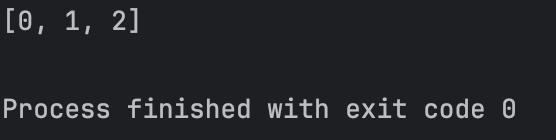
Метод retainAll оставляет в объекте intersection только те элементы, которые есть и в intersection и в hashSet2.



Следующим делом реализуем **РАЗНОСТЬ** множеств

Set<Integer> difference = new HashSet<>(hashSet);  
difference.removeAll(hashSet2);

Метод removeAll наоборот удаляет из множества difference те элементы, которые также есть в множестве hashSet2.



**УРОК 9 – МНОЖЕСТВА SET**

Мы помним, что ключи, реализованные в классах Map и Set уникальны. В случае с Map у нас не может быть два одинаковых ключа, а в случае с Set не может быть двух повторяющихся элементов.

Понятное дело, что работает это не каким-то магическим образом, а реализовано средствами языка. Как?

Когда речь идет о целых числах, то, как можно предположить, добавленный элемент сравнивается со всеми элементами этих структур. Если повторение найдено, то Map перезаписывает значение у пары по совпавшему ключу, а Set просто не добавляет новый элемент.

Но что джава делает, когда мы кладем в качестве значения в Set или ключа в Map не какой-то примитив, а какой-то объект?

На самом деле примитивом мы не можем указать тип коллекции. Это делается через классы-обертки. То есть также создаются объекты. И эти объекты проверяются на идентичность.

Делается это через контракт HashCodeEquals.Чтобы понять как это работает, передадим в качестве типа-аргумента наш собственный объект

Рядом создадим класс Person. У наших людей будет id и name. Так же сгенерируем конструктор, который будет принимать аргументы при создании объектов. И переопределим метод toString(), чтобы он выдавал такое представление о наших объектах:

"Person{" + "id=" + id + ", name='" + name + '\'' + '}';

class Person {  
 private int id;  
 private String name;  
 public Person(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
 @Override  
 public String toString() {  
 return "Person{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 '}';  
 }  
}

Теперь создадим пару объектов класса Person

Person person1 = new Person(1, "Mike");  
Person person2 = new Person(2, "Katy");

Теперь создадим по объекту класса Set и Map и установим им тип-параметр Person.

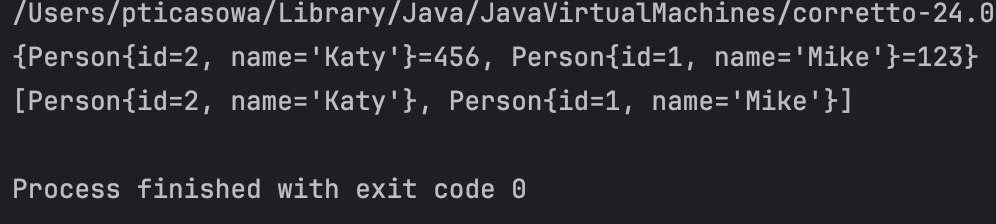
Map<Person, String> map = new HashMap<>();  
Set<Person> set = new HashSet<>();

И добавим в эти структуры объекты класса Person.

map.put(person1,"123");  
map.put(person2,"456");  
set.add(person1);  
set.add(person2);

И выведем на экран:

System.*out*.println(map);  
System.*out*.println(set);



Теперь сделаем интересную вещь:

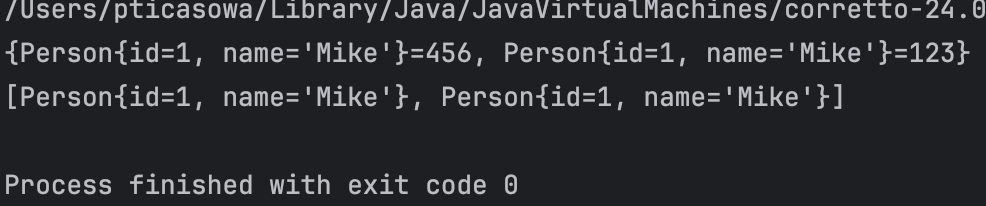
У объекта person2 поменяем поля на такие же, как у объекта person1.

Теперь у нас есть два разных объекта класса Person, но у них идентичные поля.

Person person1 = new Person(1, "Mike");  
Person person2 = new Person(1, "Mike");

И хоть это и разные объекты, их можно воспринимать как идентичные, потому, что у них одинаковое имя и id.

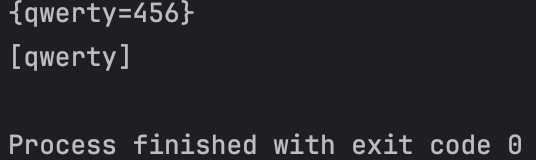
И если мы запустим программу..



.. то увидим, то Map и Set восприняли объекты как разные, хоть они и идентичны. Происходит это потому, что java сравнивает не содержимое объектов, а сами объекты.

Но почему у строк нет такого поведения?

String string1 = "qwerty";  
String string2 = "qwerty";  
Map<String, String> mapStr = new HashMap<>();  
Set<String> setStr = new HashSet<>();  
mapStr.put(string1,"123");  
mapStr.put(string2,"456");  
setStr.add(string1);  
setStr.add(string2);  
System.*out*.println(mapStr);  
System.*out*.println(setStr);



Дело в методах hashset и equals. Два эти метода изначально определены в классе Object. В этом классе находятся базовые версии:

* equals() сравнивает ссылки (т.е. ==)
* hashCode() возвращает адресоподобный хеш (обычно зависящий от объекта в памяти)

Но! Большинство стандартных классов переопределяют их

Потому что для коллекций нужно логическое равенство, а не сравнение ссылок.

Наиболее важные стандартные классы, которые переопределяют equals и hashCode:

1. Строки и числа

* String
* Integer, Long, Short, Byte
* Character
* Boolean
* Все классы-обёртки примитивов

1. Большие числа

* BigInteger
* BigDecimal

1. Дата и время (новые API)

* LocalDate
* LocalDateTime
* LocalTime
* Instant
* Duration
* Period

1. Коллекции

* ArrayList
* LinkedList
* HashSet
* TreeSet
* HashMap
* LinkedHashMap
* TreeMap

Все коллекции переопределяют equals() и hashCode() так, чтобы сравнивать *содержимое*.

|  |  |  |
| --- | --- | --- |
|  | В классе Object | В классах, где методы переопределены |
| Что делает equals() | Сравнивает ссылки | Сравнивает объекты |
| Что делает hashCode() | Возвращает адресоподобный хеш. Object гарантирует разный hashcode для двух объектов практически всегда, потому что он связан с расположением объекта в памяти. Понятное дело что у двух разных, но идентичных по содержанию объектов хэш будет разный (так как находятся они в разных местах). По сути непереопределенный hashCode является идентификатором объекта. | Возвращает хеш, сформированный из содержимого объекта. Если у класса есть свое логическое равенство equals(), то hashCode обязан быть согласован с ним.  Если a.equals(b) == true,  то a.hashCode() == b.hashCode().  Поэтому классы, которые сравнивают объекты по содержимому, а не по ссылке, обязаны вычислять hashCode тоже по содержимому. |

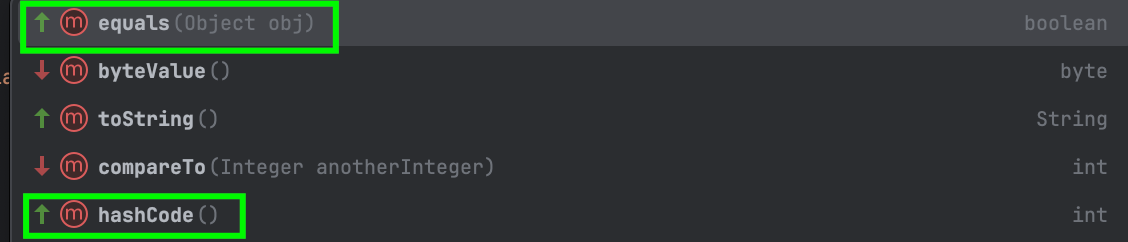
Поэтому в случае со String, ключи в паре Map и элементы в Set элементы с одинаковым содержимым расцениваются как один элемент. А случае с нашим собственным классом Person такого не происходит, потому что мы в нем не переопределяли методы equals и hashCode.

Мы можем сами вызвать эти методы у наших объектов.

Создадим объект класса Integer

Integer x = 1;

И у этого объекта мы можем вызвать методы equals и hashCode.



Оба эти метода переопределены и они возвращают уже результат сравнения самих объектов, а не ссылок на них, а также хеш кодов, основанных на их содержимом, а не на адресе в памяти.

У нашего объекта Person мы также можем вызвать эти методы, так как класс Person, как и любой другой класс Java наследуется от Person. Но они не перопределены и реузльтат выполнения методов будет результатом сравнения ссылок и хеш-кодов основанным на адресе объекта в памяти соответственно.

Чтобы исправить эту проблему, чтобы у нас правильно работали Set и Map, реализуем методы equals и hashCode в нашем классе Person. Мы мжем сами реализовать их, а можем воспользоваться шаблоном среды разработки.

Сгенерируем их с помощью среды разработки:

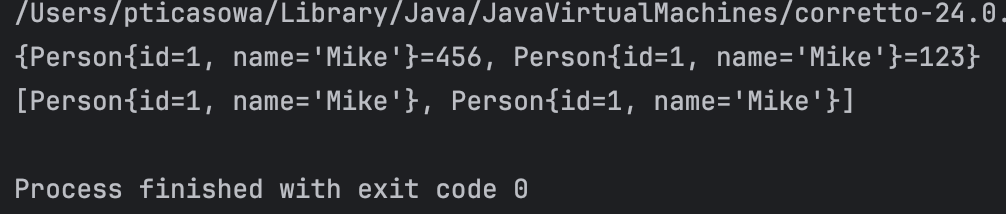
Generate - cmd+N

|  |  |
| --- | --- |
|  | Выбираем equals и hashCode |
|  | instanceof vs getClass()  Определяет:  — разрешать ли сравнение наследникам (instanceof),  — или требовать строго одного класса (getClass()).  Use getters  Использовать ли геттеры вместо полей. |
|  | Здесь выбираем по каким полям будут сравниваться объекты. Если оставим только id, то и сравниваться будут только id. |
|  | То же самое для hashcode. |
|  | equals и hashCode созданы |

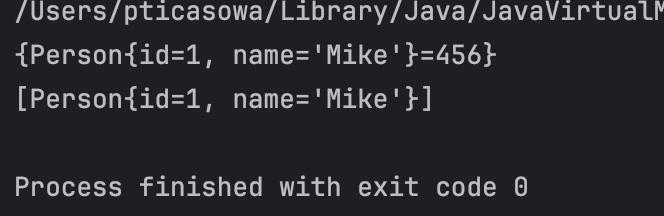
Теперь программа знает как сравнивать объекты, а не ссылки на них.

Запустим программу и посмотрим что изменилось:

Было так:



Стало так:



**УРОК 10 – КОНТРАКТЫ HASHCODE И EQUALS**

На прошлом уроке мы поняли, что если в нашем собственном классе не переопределим классы equals и hashCode, то мы будем использовать те методы equals и hashCode, которые определены в классе Object.

Зачем же использовать два метода для проверки объектов на равенство, почему мы не могли ограничиться например методом equals?

Все дело в том, что используя эти два метода, мы сравниваем объекты наиболее быстрым методом. Если б у нас не было метода hashcode и мы бы каждый раз использовали метод equals это было бы очень долго. В реализации метода equals мы проходимся по всем полям объектов и сравнить их. Когда полей у объектов не 2, не 3, а 1000, это занимает много времени.

в HashMap/HashSet hashCode используется только для быстрого сужения поиска, а не для определения равенства объектов.

Если два объекта попадают в одну корзину, HashSet НЕ считает их равными сразу.

Он делает следующее:

* Сначала сравнивает hashCode
* Если хеши разные → объекты точно разные.
* Если хеши одинаковые → возможна коллизия → нужно проверить equals()

Затем вызывается equals()

* Если equals возвращает true — объекты считаются одинаковыми
* (второй не добавляется в Set)
* Если equals возвращает false — объекты разные
* (они оба будут лежать в этой корзине)

Что такое хеширование? Хеширование - преобразование массива входных данных ***произвольной*** длины в (выходную) битовую строку ***фиксированной*** длины, выполняемое определенным алгоритмом. Функция, реализующая лгоритм называется хеш-функцией. Исходные данные называются ключом или сообщением. Результат называется хешем.

Метод hashCode и есть та самая hash функция.

Зачем все это делается? Потому, что целые числа сравнивать гораздо проще чем объекты разной длины (если мы сравниваем объекты произвольной длины, то мы должны сравнить все поля этих объектов).

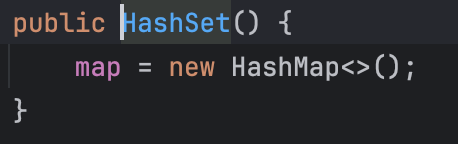
**УРОК 11 – КАК УСТРОЕН HASHCODE**

На этом уроке мы посмотрим как хеширование применяется в java коллекциях и конкретно посмотрим на реализацию класса HashMap, потому что в этом классе использование хэширования позволяет значительно ускорить работу этой коллекции.

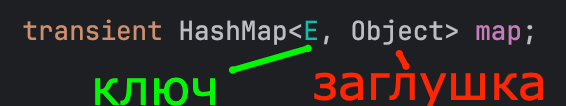
Почему мы разбираем реализацию хэшмэп, а не например хэшсет? Дело в том, что в java hashset в своей реализации использует hashmap. Почему? Потому, что hashset - это та коллекция, которая хранит в себе только уникальные значения, а в hashmap у нас уникальные ключи.

Поэтому хэшсет не имеет собственной реализации, внутри своей реализации он упирается на хэшмэп и значения в хэшсете - это на сама деле ключи в хэшмэпе.

Если мы зайдем в реализацию хэшсет, мы это увидим:

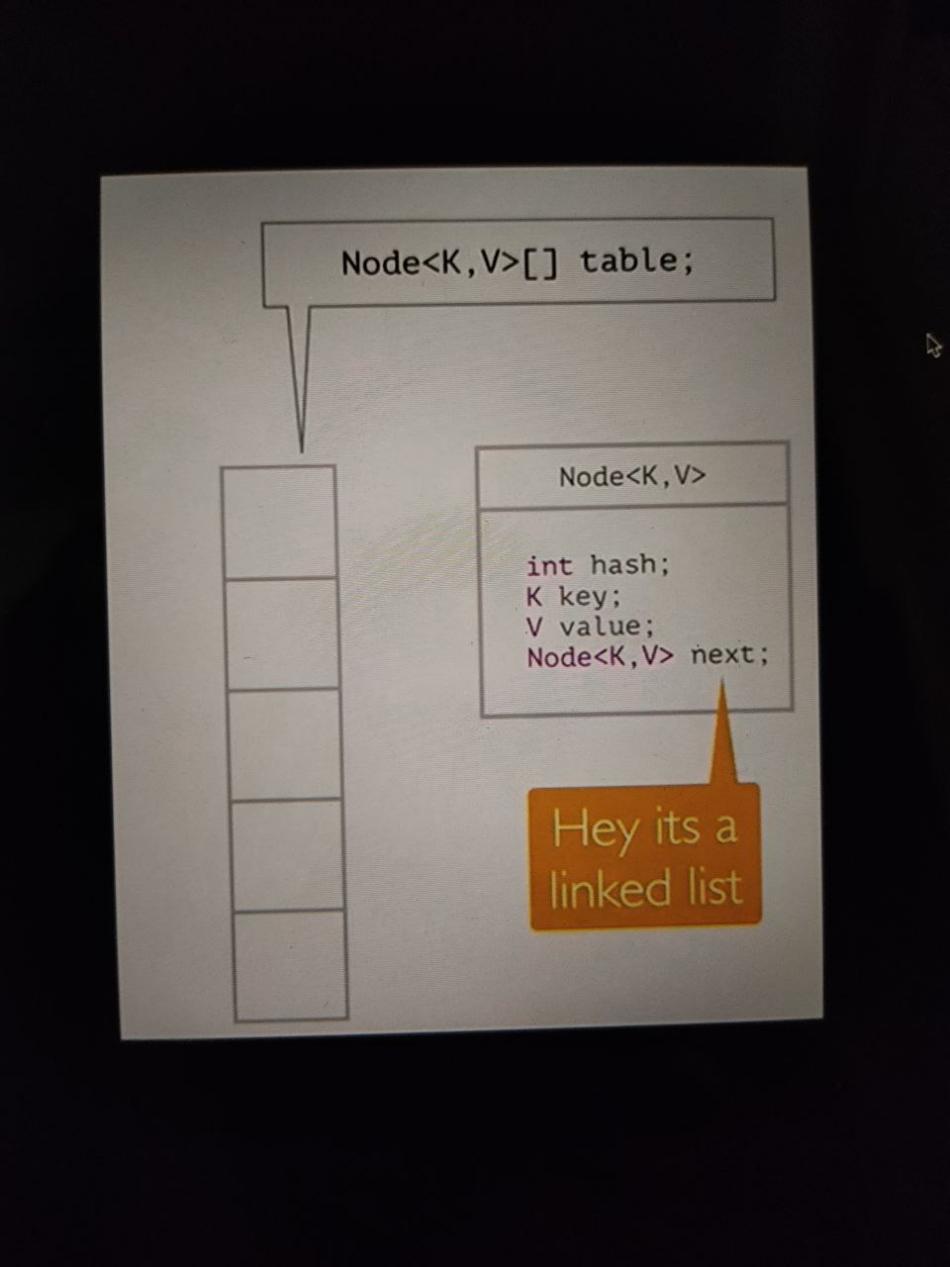


А в качестве значений ему всегда указывается класс Object, выступая своего рода заглушкой, к которой мы никогда не обратимся.



А вот у хэшмэпа реализация своя, в ней используется хэширование.

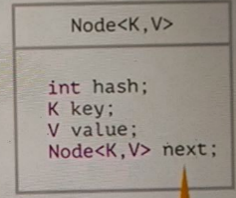
Объяснение работы хэшмэпа графически:



На этой картинке мы видим реализацию класса hashmap. Мы видим, что класс hashmap внутри себя использует массив. В массиве мы можем очень быстро обращаться к элементам по их индексу. И в каждой ячейке этого массива хранится связный список.

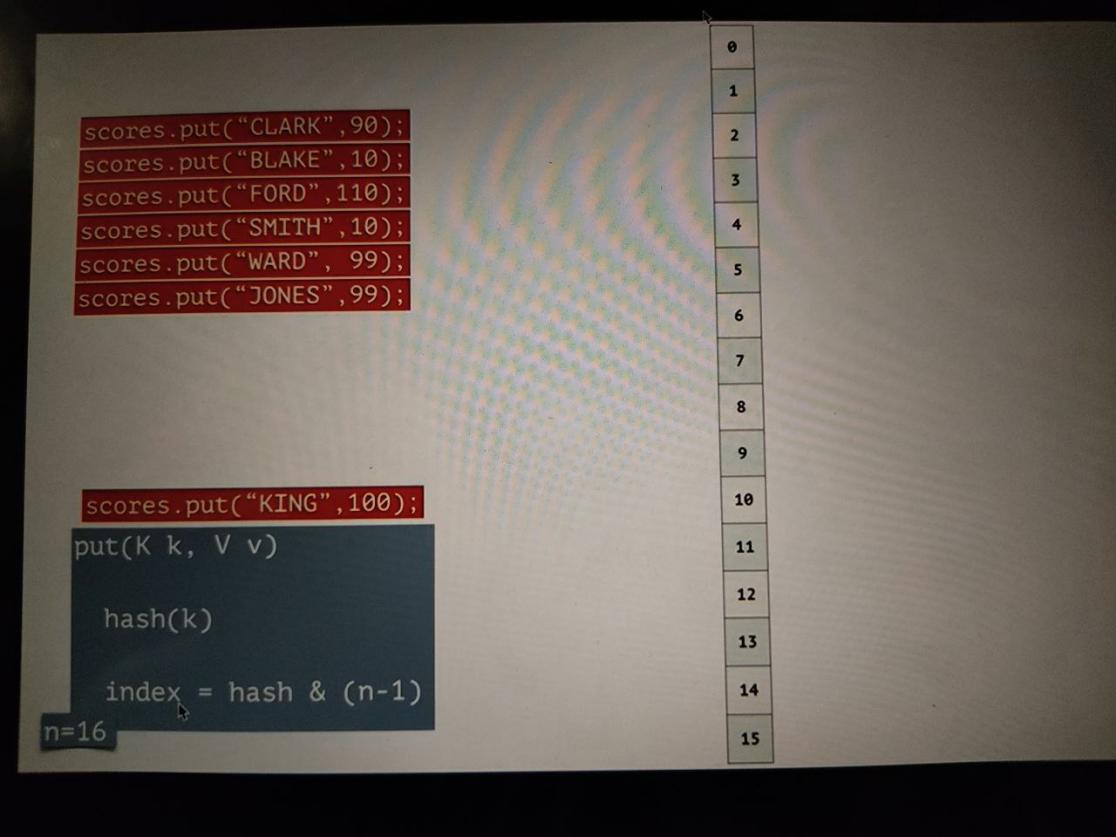
Хэшмэп в своей релизации использует массив связных списков

Мы также видим, что в каждом связном списке каждой ячейки содержится узел со структурой:



Каждый узел содержит целочиленное значение хеш, ключ, значение и ссылку на следующий узел.

Посмотрим как этот массив связных списков позволяет так быстро и эффективно выполнять операции put и get

.

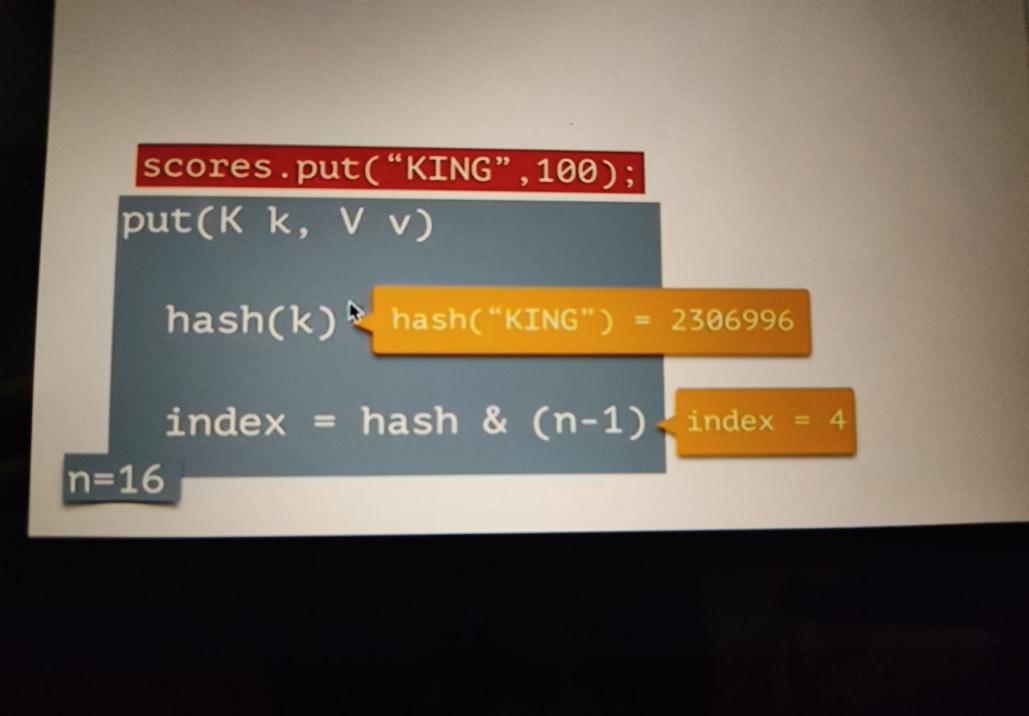
На этой картинке мы видим наш хэшмэп. Внутренний массив пока что пустой.

Теперь добавим в этот хэшмэп пару ключ-значение KING 100



Т.е. наш мэп хранит в себе в качестве ключей строки, а в качестве значений числа.

Когда мы вызовем метод put, у нашего ключа будет подсчитан хэш.

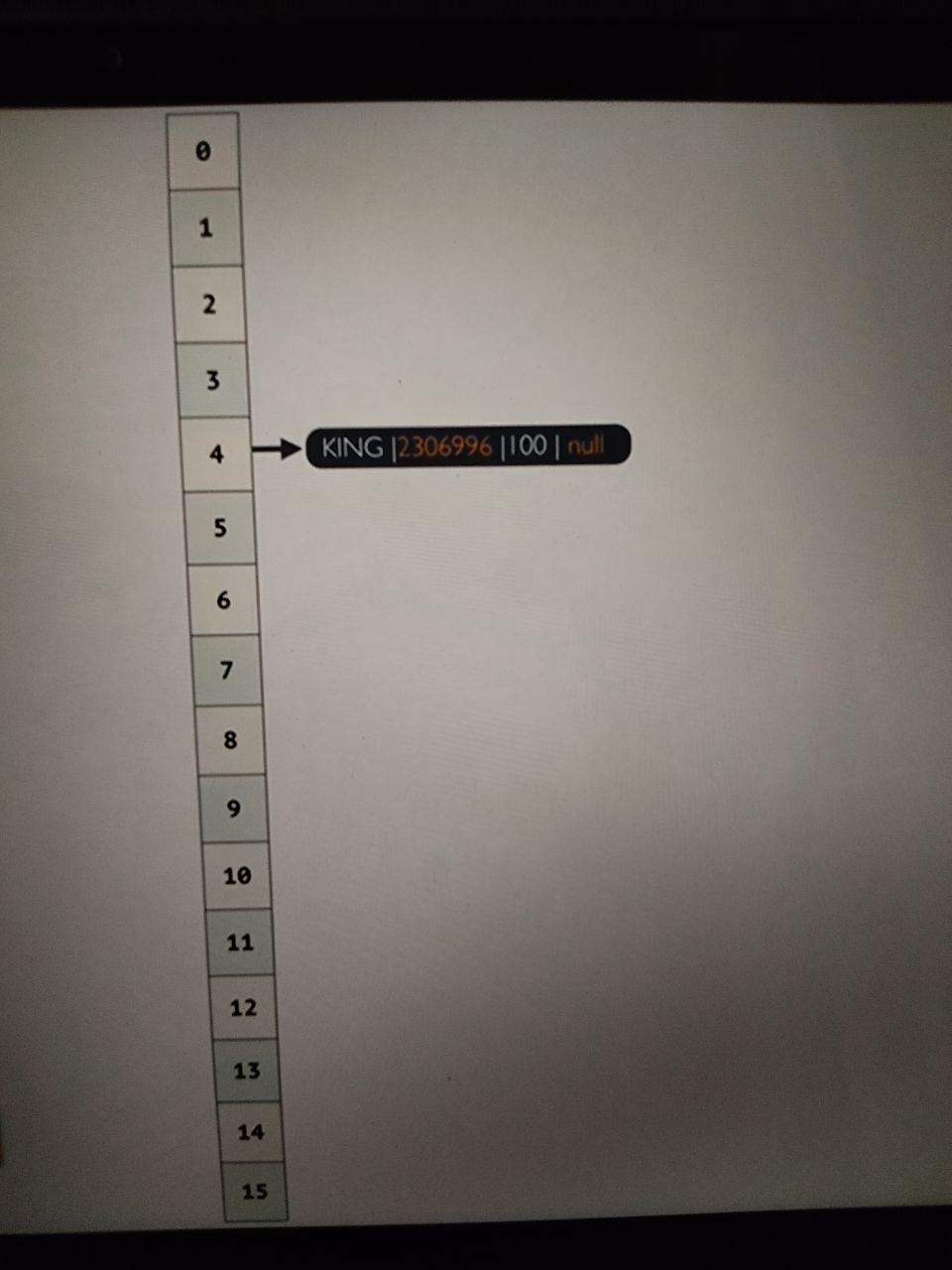


В любом классе, объекто которого мы хотим использовать в качестве ключа для Map должен быть реализован метод hashCode. Этот метод в этом случае возвращает число 2306996. Но это число не подходит в качестве индекса для нашего массива, оно слишком большое.

Но мы хотим поместить этот элементв наш массив из 16 элементов. Чтобы это сделать мы наш хэш побитово умножамем на n-1. Если абстрагироваться от побитового умножения, то это ничто иное как нахождение остатка от деления. Index в нашем случае всегда будет находиться в промежутке от 0 до 15.

\*ИЗУЧИТЬ ПОБИТОВОЕ УМНОЖЕНИЕ\*

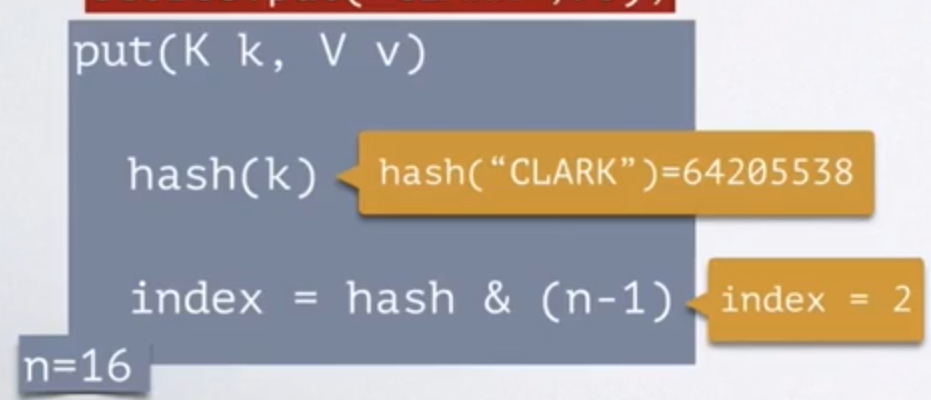
У нас получилось число 4, которое будет служить для нас индексом. Теперь когда мы добавляет элемент с ключом КИНГ, у в ячейке с индексом 4 появляется следующий узел:



Как мы видим, у этого узла нет ссылки на следующий, потому, что он пока что один.

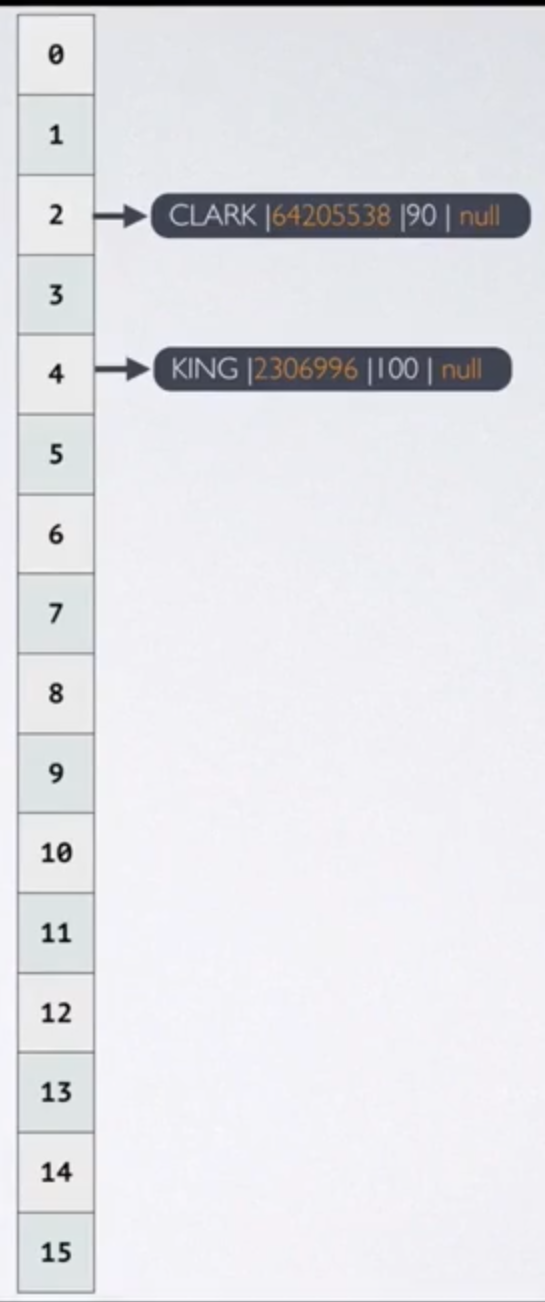
Теперь добавим следующий элемент, следующую пару ключ-значение: КЛАРК, 90





Мы посчитали хэщкод. Хэш у него 64205538.

Затем передали это число на следующую строку и побитово умножили его на n-1. Получили индекс 2. Размещаем узел в ячейку 2.



Берем следующую пару ключ значение, проворачиваем все те же действия.



Мы видим, что у нас снова индекс = 4.

В ячейке 4 у нас уже лежит некоторое значение. Поэтому мы и используем связные списки, чтобы быстро и эффективно добавлять новые узлы.

У нас просто происходит добавление новых элементов в конец нашего связного списка.

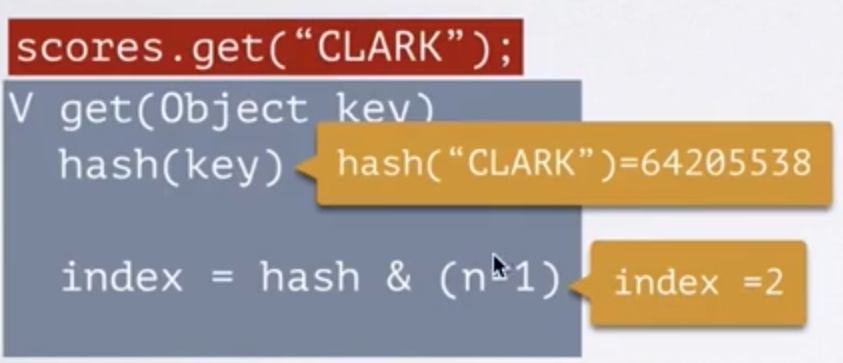


Это очень быстро, ведь в предыдущем узле мы всего лишь меняем ссылку на следующий. А в последнем элементе у нас снова ссылка на следующий = null.

По аналогии добавляем остальные элементы

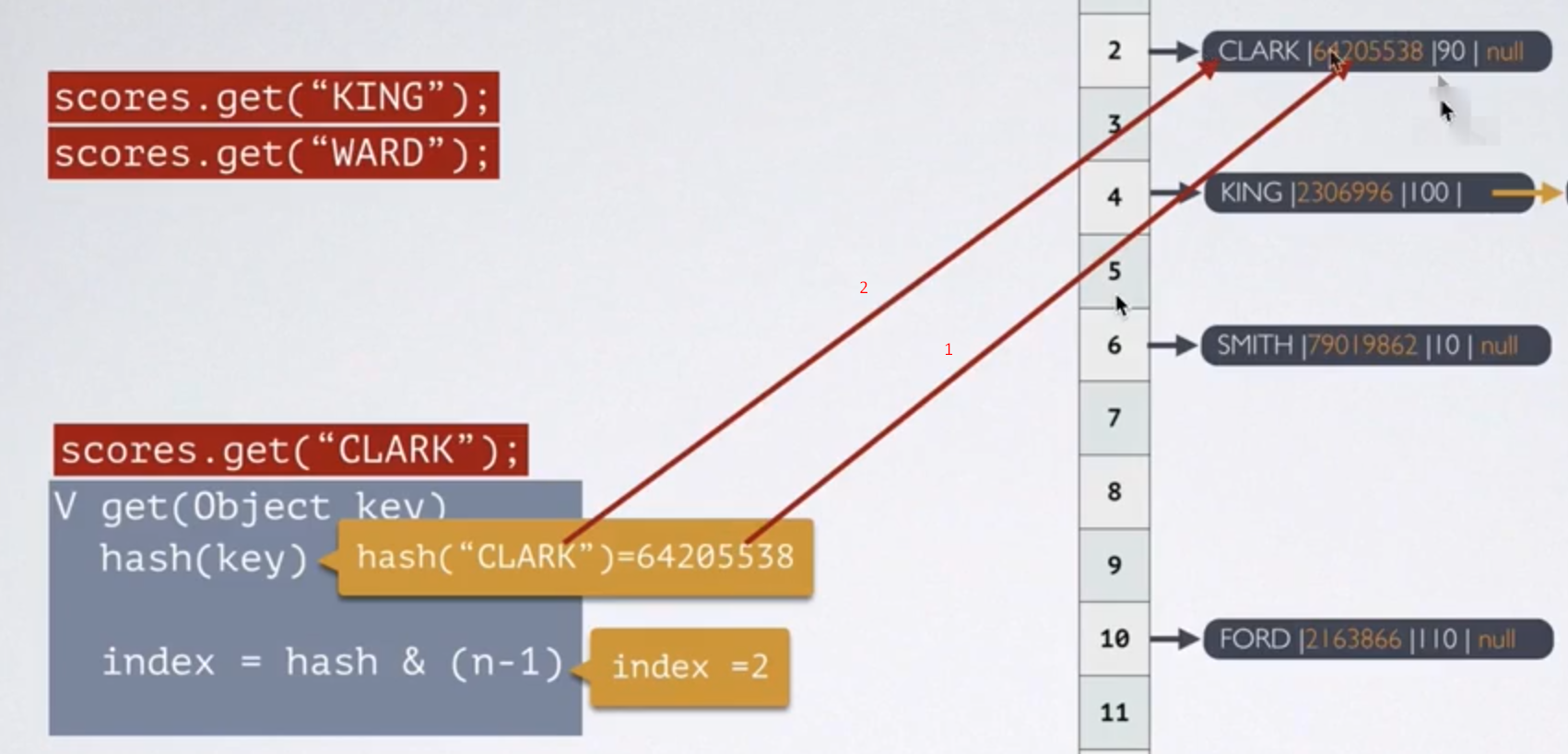


Теперь посмотрим как работает операция get. Допустим, мы хотим достать элемент с ключом CLARK.



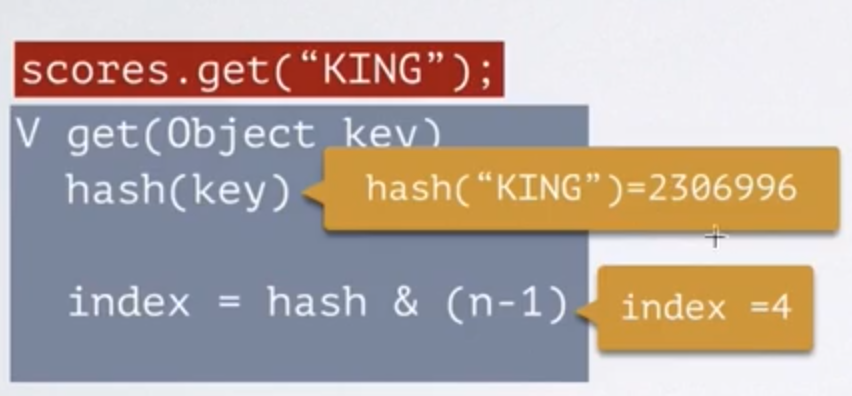
Что мы делаем? Мы высчитываем хеш ключа и индекс ячейки, в которой лежит нужный нам узел.

Сначала мы проверяем что хеш коды равные, затем проверяем что сами ключи равны и потом мы выдаем значения по этому ключу - 90.



Теперь попробуем получить значение по ключу КИНГ:

Высчитываем хеш и индекс:



Мы знаем, что где-то в ячейке 4 находится наша пара ключ-значение



Мы заходим в наш связный список и начинаем проверять равенство хешей и самих ключей.

Если хеши равны, то после них сравниваются сами ключи с помощью equals(если равнф возвращаем значение – 100, если не равны, то переходим к следующему узлу и т.д.).

Если хеш функция хорошая и коллизии встречаются редко, то операции в хэшмэпе практически не уступают по скорости операциям в массиве.

Что будет если хеш функция плохая? Ответ: Коллизии будут встречаться часто, то во первых много элементов будут попадать в одни и те же ячейки массива, что приведет к тому что хэшмэп станет больше похожим на линкедЛист.

Плюс ко всему будут частые проверки методом equals.

**УРОК 12 – ИНТЕРФЕЙС COMPARATOR**

На прошлых уроках мы разобрали важнейшие классы из Java Collections Framework и научились проверять объекты на равенство с помощью методов hashcode и equals.

На этом уроке мы рассмотрим тему сортировки объектов в Java коллекциях.

Бывают такие случаи, когда мы хотим, чтобы объекты, которые хранятся в нашей Java коллекции были отсортированы по какому-то принципу. На этом уроке мы научимся сортировать объекты внутри Java коллекции.

Стоит отметить, что в этих уроках мы будет использовать те коллекции, которые сохраняются в себе порядок. Например, мы не будем использовать класс hashset и hashmap, потому что внутри них нет сортировочного механизма и не гарантирует никакого порядка объектов.

Поэтому в этих уроках мы будем использовать классы, реализующие интерфейс Лист (ArrayList, LinkedList и т.д.). А также сортировку внутри себя поддерживают такие классы как TreeSet и TreeMap.

Создадим пару листов и отсортируем объекты в этих листах.

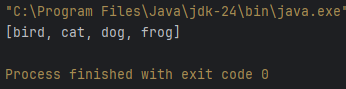
Первый лист у нас будет хранить объекты типа String. Добавим несколько объектов

public class Test {  
 public static void main(String[] args) {  
 List<String> animals = new ArrayList<>();  
   
 animals.add("dog");  
 animals.add("cat");  
 animals.add("frog");  
 animals.add("bird");  
 }  
}

Вызовем статический метод sort класса Collection и передадим ему в качестве аргументов наш ArrayList animals. Этот метод сортирует объекты согласно естественному порядку, который определен на объектах этого листа. О естественном порядке поговорим в следующих уроках, а пока запустим этот метод

Collections.*sort*(animals);

System.*out*.println(animals);

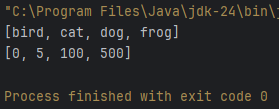


Как мы видим, порядок объектов в нашем листе поменялся, теперь они отсортированы по лексикографическому порядку.

Теперь создадим лист интеджеров и также отсортируем, здесь естественный порядок будет лучше виден.

List<Integer> numbers = new ArrayList<>();  
  
numbers.add(5);  
numbers.add(0);  
numbers.add(500);  
numbers.add(100);

Collections.*sort*(numbers);  
System.*out*.println(numbers);



Видим по возрастанию стоящие числа. Этот порядок называется естественным или по умолчанию. На англ. – natural order.

Но что если этот порядок нам не подходит? Например мы хотим, чтобы сортировка чисел происходила по убыванию, а не по возрастанию. Или мы хотим, чтоб наши строки сортировались не лексикографически, а по длине.

Для таких случаев и существует интерфейс Comparator. Мы можем реализовать интерфейс и внутри указать ту логику по которой мы хотим сортировать наши объекты и затем применить эту логику к нашим коллекциям. Теперь посмотрим как это сделать.

Чтобы добавить свою логику сортировки объектов мы должны добавить свой класс и в этом классе мы должны реализовать интерфейс Comparator. У этого интерфейса есть всего лишь одни метод и когда класс его реализует, этот класс становится компаратором. И объект этого класса мы можем передавать в метод sort в качестве второго аргумента. И после этого объекты в нашей коллекции будут сортироваться по той логике, которая задана в компараторе.

Создадим новый класс, назовем его StringLengthComparator.

Интерфейс Comparator параметризованный, поэтому мы должны указать тип-аргумент. В нашем случае это String. И реализуем метод compare:

class StringLengthComparator implements Comparator<String> {  
  
 @Override  
 public int compare(String o1, String o2) {  
 return 0;  
 }  
}

Именно в этом методе мы должны описать всю логику сравнивания двух строк между собой.

Важно отметить то, что в методе compare существует своя конвенция. Метод возвращает целое число и по этому целому числу Java понимает, что объект 1-ый большое 2-ого, или наоборот, или они равны. И для того, чтобы Java это поняла, существует своя конвенция:

/\*  
 o1 > o2 => 1;  
 o1 < o2 => -1;  
 01 == 02 => 0;  
  
compare(2,1) => 1;  
compare(1,2) => -1;  
compare(1,1) => 0;  
  
 \*/

Если первый переданный объект больше второго, то возвращается 1;

Если наоборот, то -1;

Если объекты равны, то возвращается 0.

Теперь то, что нам осталось реализовать – если длина первой строки больше длины второй -> возвращаем 1. Если наоборот, то -1. Если длины равны, то 0.

@Override  
public int compare(String o1, String o2) {  
 if (o1.length()>o2.length()){  
 return 1;  
 }  
 else if (o1.length()<o2.length()){  
 return -1;  
 }  
 else{  
 return 0;  
 }  
}

Передадим в метод sort объекта класса StringLengthComparator. Для наглядности добавим побольше элементов в наш линкедЛист.

animals.add("dog");  
animals.add("bird");  
animals.add("elephant");  
animals.add("mouse");  
animals.add("giraffe");  
animals.add("cat");  
animals.add("frog");

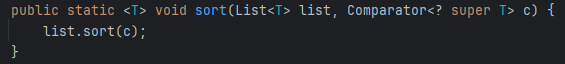
StringLengthComparator stringLengthComparator = new StringLengthComparator();

Collections.*sort*(animals, stringLengthComparator);



Как мы разбирали ранее, в метод sort может быть передано два аргумента. Первый объект - это что мы сортируем, а второй – это каким образом мы сортируем первый объект.

Если перейдем в реализацию метода sort в классе Collections, то увидим в параметрах переменные типа List и Comparator.



Теперь лексикографический порядок не работает, работает сортировка по тому порядку, который мы создали сами.

Теперь сделаем то же самое для List numbers. Допустим мы хотим сортировать наши числа в убывающем порядке.

Для этого мы должно так же реализовать свой собственный Comparator, но на этот раз с параметризацией интерфейса типа Integer.

class BackwardsIntegerComparator implements Comparator<Integer> {  
 public int compare(Integer o1, Integer o2) {  
 if (o1>o2){  
 return -1;  
 }  
 else if (o1<o2){  
 return 1;  
 }  
 else{  
 return 0;  
 }  
 }  
}

Почему в условиях if теперь -1 находится при 01>02? Потому что теперь нам нужно чтобы в начале были самые большие числа.

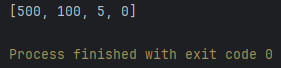
1 значит элементы меняются, 0 и -1 значит, что остаются на месте. Если мы сравниваем 5 и 0, o1 > o2 => -1, элементы остаются на месте. Сравниваем 0 и 500. , o1 > o2? Нет , o1 < o2? Да => 1 – меняем элементы и т.д.

Запомнить легко.

Если мы возвращаем -1, то большее в паре число будет стремиться к началу отсортированного списка (ближе к -1).

Если возвращаем 1, то большее в паре число будет стремиться к концу (к 1).

Collections.*sort*(numbers, new BackwardsIntegerComparator()); //мы создали объект прямо здесь, ведь использовать его где либо еще не намерены, так можно.  
System.*out*.println(numbers);



Эти сравнение работают до тех пор, пока во всем списке не встретится ни одной перестановки. Если перестановок нет, значит все уже стоит на своих местах.

Еще стоит поговорить о том, что реализовывать класс рад одного того, чтобы реализовать один только интерфейс и метод слишком громоздко.

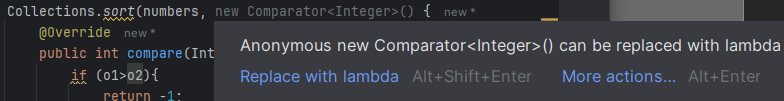
Поэтому чтобы не создавать лишний класс, который будет использоваться только один раз, мы можем использовать анонимные классы, пройденные раннее. ПОВТОРИТЬ АНОНИМНЫЕ КЛАССЫ! – Java для начинающих.

Выглядеть это добро будет так:

List<Integer> numbers = new ArrayList<>();  
numbers.add(5);  
numbers.add(0);  
numbers.add(500);  
numbers.add(100);  
Collections.*sort*(numbers, new Comparator<Integer>() {  
 @Override  
 public int compare(Integer o1, Integer o2) {  
 if (o1>o2){  
 return -1;  
 }  
 else if (o1<o2){  
 return 1;  
 }  
 else{  
 return 0;  
 }  
 }  
});  
System.*out*.println(numbers);

Мы создаем анонимный класс из интерфейса Comparator и тут же реализуем метод этого интерфейса.

Если мы наведемся на код создания анонимного класса, то увидим, что среда разработки предлагает нам заменить этот класс лямбдой.



Лямбды будут изучены в дальнейших уроках. Но лямбды позволяют еще больше сократить код

И закончим урок тем, что создадим еще один Comparator для сравнения наших собственных объектов.

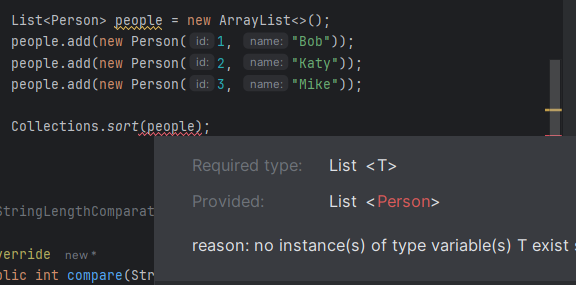
Реализуем класс Person и предположим, что у каждого Person будет id и name и создадим методы toString, getId и setId

class Person{  
 private int id;  
 private String name;  
 public Person( int id, String name) {  
 this.name = name;  
 this.id = id;  
 }  
 @Override  
 public String toString() {  
 return "Person{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 '}';  
 }  
 public int getId() {  
 return id;  
 }  
}

Теперь создадим List, который будет содержать объекты класса Person и добавим несколько последних.

List<Person> people = new ArrayList<>();  
people.add(new Person(1, "Bob"));  
people.add(new Person(2, "Katy"));  
people.add(new Person(3, "Mike"));

Теперь добавим метод sort и попробуем отсортировать объекты.



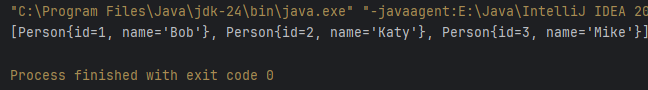
И сразу получаем ошибку, так как мы в нашем классе Person не задали естественного порядка. В случае со строками (лексикографический порядок) и целыми числами (по возрастанию), естественный порядок уже задан.

Задавать естественный порядок мы научимся в следующем уроке. А пока мы в методе sort зададим Comparator, который на себя возьмет функцию установки порядка.

Пусть мы хотим сортировать элементы коллекции по возрастанию номера id.

Collections.*sort*(people, new Comparator<Person>() {  
 @Override  
 public int compare(Person o1, Person o2) {  
 if (o1.getId()>o2.getId()){  
 return 1;  
 } else if (o1.getId()<o2.getId()){  
 return -1;  
 } else {  
 return 0;  
 }  
 }  
});

System.*out*.println(people);



**УРОК 13 – ИНТЕРФЕЙС COMPARABLE**

На прошлом уроке мы научились сортировать объекты в коллекциях, поставляя свой собственный comparator в метод sort для того чтобы определить порядок в нашей коллекции.

Мы заметили, что для сортировки, например, строк и целых чисел нам не обязательно было поставлять свой компаратор. В этих классах объявлен естественный порядок, а в наших же классах этого естественного порядка нет.

И на этом уроке мы дадим объекту нашего собственного класса естественный порядок.

Для этого создадим класс Person.

У нашего Person будут поля id и name, конструктор, который принимает значения при создании объектов и метод toString и создадим методы hashCode и Equals. Потому, что в ходе этого урока объекты будут добавляться в Сет. Также добавим геттеры для id и name (естественный порядок сравнения будет по id и name)

Создадим Лист и Сет, которые параметризованы типом Person. Сет на это траз возьмем ТриСет, который сортирует элементы внутри себя, в отличие от ХэшСет.

List<Person> peopleList = new ArrayList<>();  
Set<Person> peopleSet = new TreeSet<>();

Теперь создадим отдельный метод, которому в аргумент будет передаваться коллекция и он будет добавлять элемент в эту коллекцию.

Collection – самый верхний предок над всеми коллекциями и за счет полиморфизма наш метод будет принимать любые элементы, являющиеся коллекциями (т.е. реализующие интерфейс Collection)

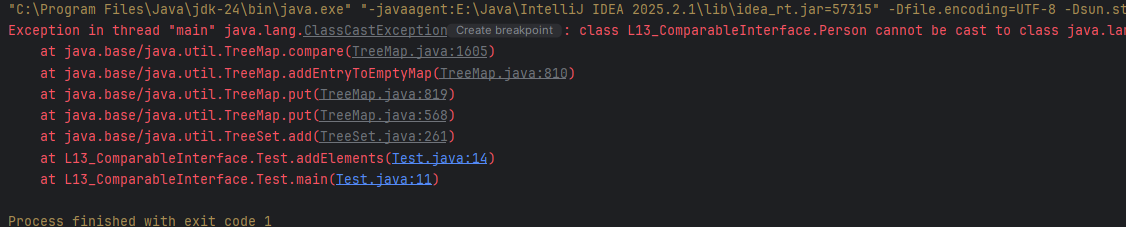
private static void addElements(Collection collection){  
 collection.add(new Person(1,"Bob"));  
 collection.add(new Person(2,"Tom"));  
 collection.add(new Person(3,"Katy"));  
 collection.add(new Person(4,"George"));  
}

И вызовем этот метод из метода main, передав ему в качестве аргумента две наши коллекции. В конце все выводим на экран.

*addElements*(peopleList);  
*addElements*(peopleSet);

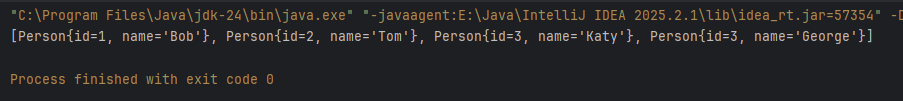
System.*out*.println(peopleList);  
System.*out*.println(peopleSet);

Запустим программу и увидим:



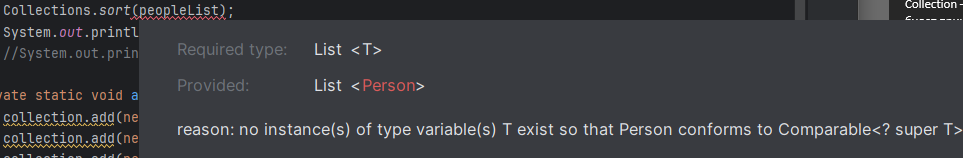
Мы получаем ошибку и видим, что TreeSet у нас хочет отсортировать объекты, а сортирует он их по естественному порядку, а в классе Person не задан естественный порядок. Как следствие TreeSet не знает как добавлять элементы, потому, что он не знает как их сортировать.

Если мы временно закоментируем TreeSet и выполним все действия только с peopleList, то получим:



Видим, что все успешно добавилось.

Если мы попробуем отсортировать объект Lista, без добавленного нами вручную компаратора, то мы также получим ошибку:

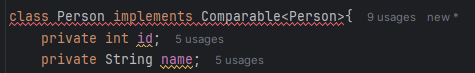


Ситуация похожая с TreeSet. Java по прежнему не понимает как сортировать элементы в peopleList без заданного порядка.

Чтобы java смогла сортировать объекты, нам нужно реализовать еще один метод. Этот метод называется compareTo.

Чтобы java понимала, что в классе существует порядок сортировки, в этом классе нужно реализовать интерфейс Comparable.

В прошлом уроке у нас был интерфейс Comparator – дословно в переводе “сравниватель”. Теперь у нас интерфейс Comparable – то бишь “сравниваемый”.



Сигнатура метода светится красным. Реализуем единственный метод этого интерфейса toCopmare()

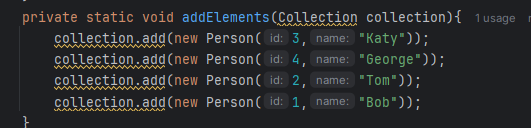
@Override  
public int compareTo(Person o) {  
 return 0;  
}

Порядок сравнения мы будем описывать в этом методе compareTo.

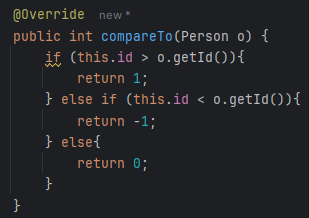
Нужно обратить внимание, что здесь в параметрах у нас только один объект класса Person, потому что метод вызывается от какого другого объекта, по типу: obj1.compareTo(obj2), в отличие от метода compare, где это был метод-наблюдатель, не вызывающийся не от какого объекта.

Теперь на нашем классе Person создадим естественный порядок. Сделаем его таким же, как и в нашем предыдущем уроке, когда вы сравнивали объекты по их id.

* + - 1. Перепутаем значения, чтобы было видно сортировку.



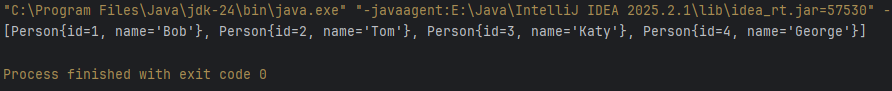
* + - 1. Напишем тело метода:



P.S. – то, что тут геттеры а не просто id будет объяснено в конце урока.

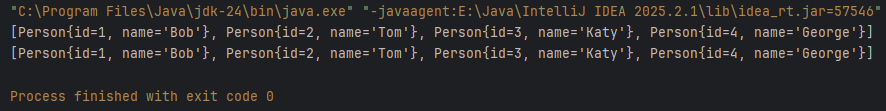
Первый элемент у нас this, потому что мы обращаемся к объекту, от которого и вызывается этот метод.

* + - 1. Запускаем программу



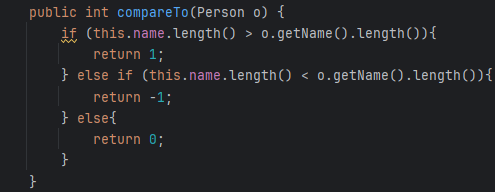
Теперь раскоментируем TreeSet:

*addElements*(peopleList);  
*addElements*(peopleSet);  
Collections.*sort*(peopleList);  
  
System.*out*.println(peopleList);  
System.*out*.println(peopleSet);



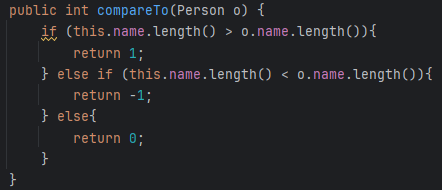
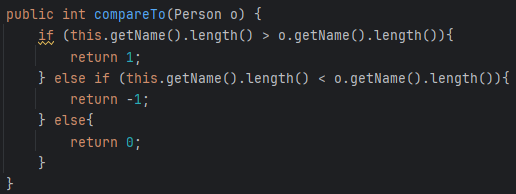
Все работает, теперь у нас есть естественный порядок по возрастанию id.

Теперь сделаем естественный порядок по длине имени name



ВАЖНО: Этот метод мы переопределяем внутри класса Person, это говорит о том, что нам необязательно обращаться к геттерам в этом случае. Но Алишев здесь вставляет и геттеры и name без геттеров. Предполагаю, что он всегда использует геттеры по умолчанию, потому что с ними точно не ошибешься, и просто запутался.

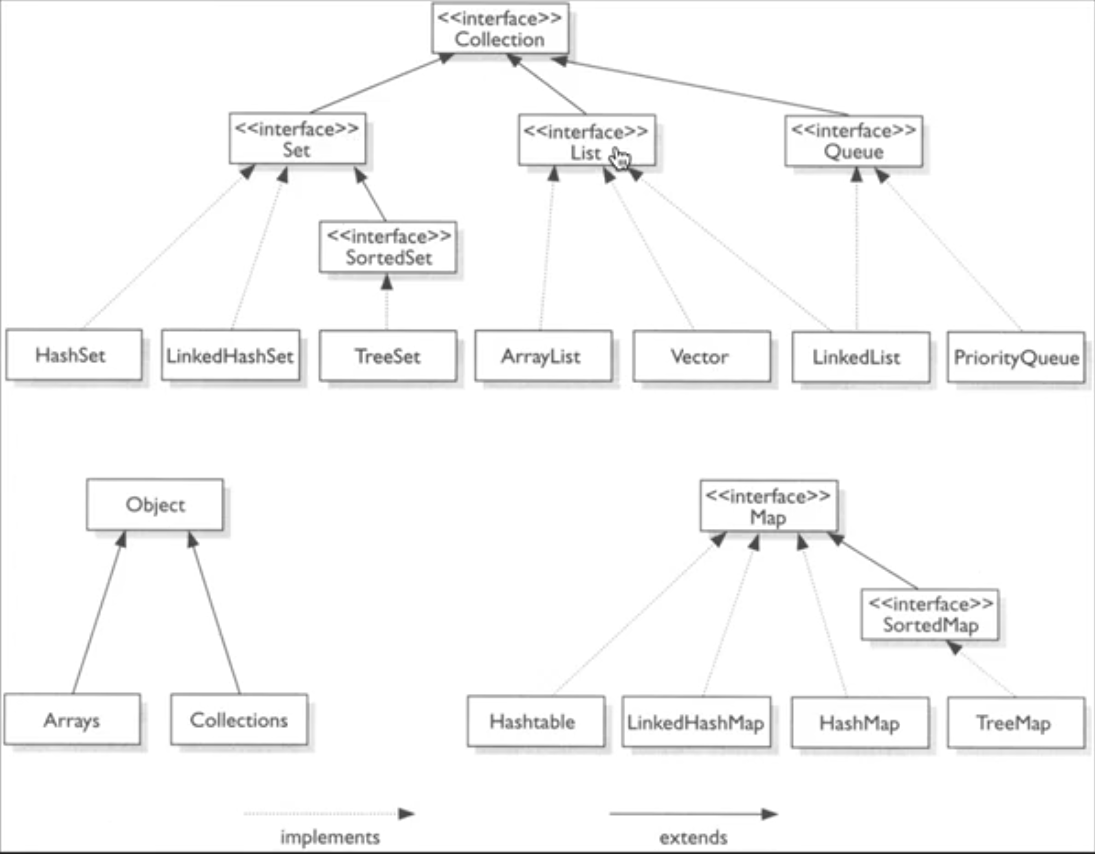
Т.е. точно также здесь будут работать и такие методы:

Аналогично и с методом с id.

**УРОК 14 – ИНТЕРФЕЙС QUEUE**

Мы прошли довольно долгий путь и изучили множество классов в java collection framework. Мы изучили классы, реализующие интерфейс Set, интерфейс List, интерфейс Map, теперь изучим интерфейс Queue.



Queue переводится с английского как очередь и эта Java коллекция реализует очередь. Т.е. мы сможем помещать объекты в нашу очередь и вытаскивать их оттуда в том порядке, в котором мы их туда поместили – FIRST IN FIRST OUT (FIFO).

Такая структура полностью повторяет обычную очередь куда угодно: в магазине, в кабинет к врачу и т.д.

В программах такая структура данных часто используется, например, при многопоточности (о многопоточности будет в будущих уроках), когда много потоков обращаются к одному ресурсу, то мы должны выстраивать какую-то очередь между этими потоками.

На диаграмме мы видим, что интерфейс Queue реализует два класса: LinkedList и PriorityQueue. На самом деле интерфейс Queue реализует намного больше классов, и об одном из них мы поговорим в этом уроке, но сейчас мы поговорим об реализации интерфейсов Queue классом LinkedList. Этот класс мы уже использовали, но теперь мы используем его в качестве реализации интерфейса Queue. О классе PriorityQueue в этом уроке мы говорить не будем, это достаточно специфическая вещь, то там нет ничего сложного, если интересно можно посмотреть информацию о нем в Google (пробью в чатгпт).

Создаем класс Person, ограничимся в нем только полем id, создаем конструктор, геттер и метод toString.

Теперь создадим несколько людей

public class Test {  
 public static void main(String[] args) {  
 Person person1 = new Person(1);  
 Person person2 = new Person(2);  
 Person person3 = new Person(3);  
 Person person4 = new Person(4);  
  
 }  
}  
class Person{  
 private int id;  
  
 public Person(int id) {  
 this.id = id;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 @Override  
 public String toString() {  
 return "Person{" +  
 "id=" + id +  
 '}';  
 }  
}

Теперь создадим очередь. Создадим объект класса LinkedList. На этот объект будет указывать переменная типа Queue, параметризованная типом Person.

Queue<Person> people = new LinkedList<>();

Людей в очереди мы расставим не по порядку.

Чтобы добавлять элементы в queue, существует два метода: add и order..

Для получения элементов из очереди существует также два различных метода: poll и remove.

Это такие пары методов (add, remove и offer,poll) и в них есть некоторые различия, но об этом позже.

Расставим людей в очереди в таком порядке: 3, 2, 4, 1

people.add(person3);  
people.add(person2);  
people.add(person4);  
people.add(person1);

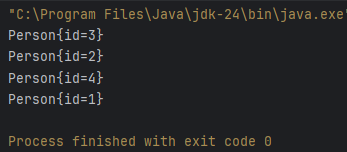
Пройдемся по нашей очереди и будем выводить каждого человека на экран с помощью цикла for each:

for(Person person : people){  
 System.*out*.println(person);  
}

Кстати, если в цикле только одна строка, то можно писать без фигурных скобок.

for(Person person : people)  
 System.*out*.println(person);

Запустим программу:



Все работает, FIFO соблюдается, кто первый зашел, тот первым и выходит.

Это самое базовое, что нужно знать об очередях, дальше мы поговорим о другой реализации Queue, которая называется ArrayBlockingQueue.

Чтобы увидеть этот класс, зайдем на страничку интерфейса Queue в Java спецификации



В All Known Implementing Classes мы видим все классы, которые реализуют интерфейс Queue:

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

Среди этих классов мы видим наш LinkedList, PriorityQueue, который мы видили на диаграмме. И кроме них тут есть полезный класс, который называется ArrayBlockingQueue. Алишев считает этот класс очень полезным, особенно он станет очень полезным ,когда мы будет говорить про многозадачность, потому, что этот класс часто используется когда у нас есть много потоков и мы ограничиваем доступ к общему ресурсу.

Разберем этот класс в нашем примере. Вместо LinkedList укажем класс ArrayBlockingQueue и тут в отличие от LinkedList мы должны указать максимальный размер нашей очереди.

В случае же с LinkedList мы могли помещать в очередь сколько угодно объектов, пока у нас не закончится память на накопителе.

В ArrayBlockingQueue мы обязаны указать число и это число является максимальным размером нашей очереди.

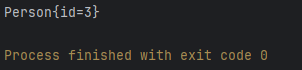
Queue<Person> people = new ArrayBlockingQueue<>(10);

И это часто бывает полезно, когда мы не хотим, чтобы наша очередь была бесконечно большой, мы должны использовать не LinkedList, а ArrayBlockingQueue.

Теперь разберем метод remove

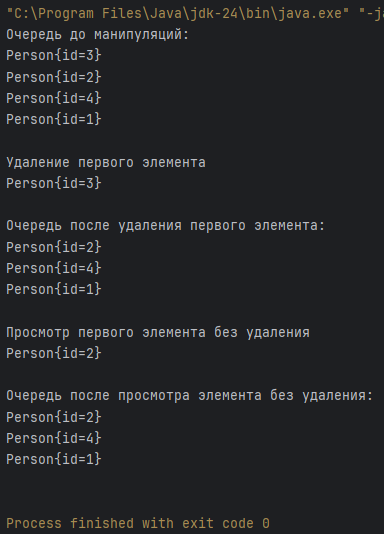
System.*out*.println(people.remove());

Этот метод удаляет из очереди первый элемент



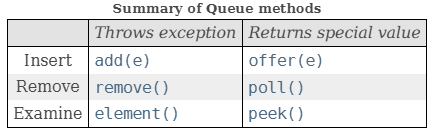
Рассмотрим еще один метод – peek. Этот метод служит для того, чтобы посмотреть на первый элемент в очереди, но не удалять его.

System.*out*.println(people.peek());



Разберем отличия между add remove и offer poll. Самый хороший способ разобраться в этом – заглянуть в спецификацию Queue.

И тут в самом начале описана их разница



По функционалу – ничем не отличаются. Add И offer добавляют элемент в очередь, remove и poll удаляют элемент из очереди. Element и peek показывает первый элемент очереди без удаления.

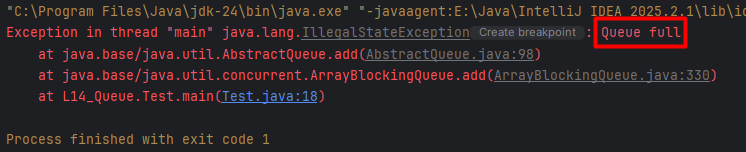
Отличие в том, что первая группа выбрасывает исключения, вторая возвращает специальные значения. Разберем на примере между add и offer.

Уменьшим размер нашей очереди до 3, все лишнее уберем.

Queue<Person> people = new ArrayBlockingQueue<>(3);  
people.add(person3);  
people.add(person2);  
people.add(person4);  
people.add(person1);

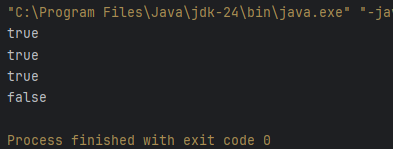
В очередь, максимальный размер которой = 3, мы добавляем 4 элемента.

Последний 4-ый метод выбросит исключение



Теперь попробуем добавить элементы не с помощью метода add, а с помощью метода offer и выведем выполнение всех методов на экран

System.*out*.println(people.offer(person4));  
System.*out*.println(people.offer(person3));  
System.*out*.println(people.offer(person1));  
System.*out*.println(people.offer(person2));



Мы видим, что первые 3 раза метод нормально отработал и вернул true, а последний раз вернул false, говоря нам о том, что элемент не добавился в очередь. Исключение при этом не выбросилось

Для всех остальных пар методов действует такая же логика.

Для чего вообще городить целый интерфейс Queue и классы, реализующие его, если с этим и так справляется класс LinkedList интерфейса List?

🔥 Краткий ответ

Интерфейс Queue нужен потому, что:

* очередь — это поведение, а не структура данных,
* LinkedList — плохая и медленная очередь,
* существуют разные типы очередей (ограниченные, приоритетные, блокирующие, lock-free, concurrent…),
* интерфейс позволяет заменить реализацию очереди, не переписывая код.

📌 Почему LinkedList НЕ подходит как очередь

Да, LinkedList может работать как очередь:

list.offer();  
list.poll();

Но есть проблемы:

❌ 1) Избыточный функционал

LinkedList — это двусвязный список, и в нём сотни операций, которые в очереди вообще запрещены:

* get(index)
* add(index)
* remove(index)
* listIterator()
* addAll()
* reverse traversal
* вставка в середину структуры

Очередь не должна позволять ничего, кроме:

* добавить в конец
* извлечь из начала

LinkedList — слишком тяжёлая структура, чтобы быть очередью.

❌ 2) Неэффективно по памяти

LinkedList хранит по 2 ссылки на каждый элемент (prev и next).

* объект Node требует выделения памяти.

Очереди вроде ArrayDeque намного более компактны.

ВАЖНО! Что такое Deque? – Double Ended Queue.

❌ 3) LinkedList медленнее, чем ArrayDeque

Это важно.

Операции offer, poll в ArrayDeque работают быстрее, чем в LinkedList, потому что:

* у LinkedList каждый элемент — отдельный объект в памяти → плохая кешируемость CPU
* у ArrayDeque элементы идут подряд → работа с CPU cache в 10+ раз быстрее

Поэтому Java документированно говорит:

ArrayDeque is likely to be faster than LinkedList when used as a queue.

ОТЛИЧИЕ LINKEDLIST ОТ ARRAYDEQUE

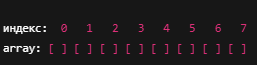
|  |  |
| --- | --- |
| 🟥 LinkedList (двусвязный список)  Хранит элементы в отдельных объектах Node:    Каждый Node — это отдельный объект в куче.  Структура:   * value * ссылка на prev * ссылка на next | 🟩 ArrayDeque (кольцевой массив)  Хранит элементы в массиве Object[], и использует два индекса:    Пример:    Если место кончится — массив увеличивается и перестраивается (как в ArrayList). |

Подробнее о ArrayDeque

**1. ЧТО ТАКОЕ КОЛЬЦЕВОЙ МАССИВ?**

Это обычный массив, но указатели head и tail могут «обегать круг».

Простой пример массива из 8 элементов:

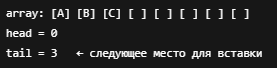


Пусть:

* head — где лежит первый элемент очереди
* tail — куда положим следующий элемент

**2. ДОБАВЛЕНИЕ ЭЛЕМЕНТА (OFFER)**

Представим, мы добавляем элементы A, B, C:



**3. УДАЛЕНИЕ ЭЛЕМЕНТА (POLL)**

Если удаляем элемент A:

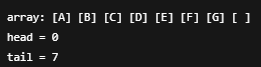


head просто сдвигается вперёд.

**4. «КОЛЬЦЕВОЙ» МОМЕНТ**

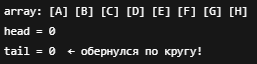
Когда хвост достигает конца массива:

Представь, что добавили элементы до конца массива:



Добавляем элемент H:

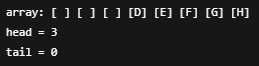
tail оборачивается в начало массива:



Если теперь удалить несколько элементов:



Получаем:



Очередь работает как круг вокруг массива.

**5. ЧТО ПРОИСХОДИТ ПРИ ПЕРЕПОЛНЕНИИ?**

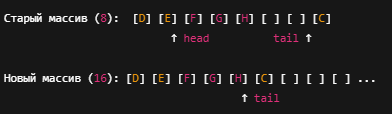
Когда:

* массив полностью заполнен
* tail догоняет head

ArrayDeque удваивает размер массива:

Был 8 → станет 16.

Элементы перекладываются так, чтобы порядок сохранялся:



Resize происходит редко → амортизированная сложность O(1).

**7. ПОЧЕМУ ЭТО ТАК БЫСТРО?**

✔ 1. Все элементы в одном куске памяти

CPU считывает их пачками → очень мало cache miss.

✔ 2. Нет объектов Node

У LinkedList каждый элемент — отдельный объект → очень дорого по памяти.

✔ 3. Нет моделирования индексов (как в LinkedList)

Всё работает через битовую арифметику, невероятно быстро.

✔ 4. Хвост и голова «ходят по кругу»

Отсюда минимальные перемещения указателей.

Итог в 5 словах

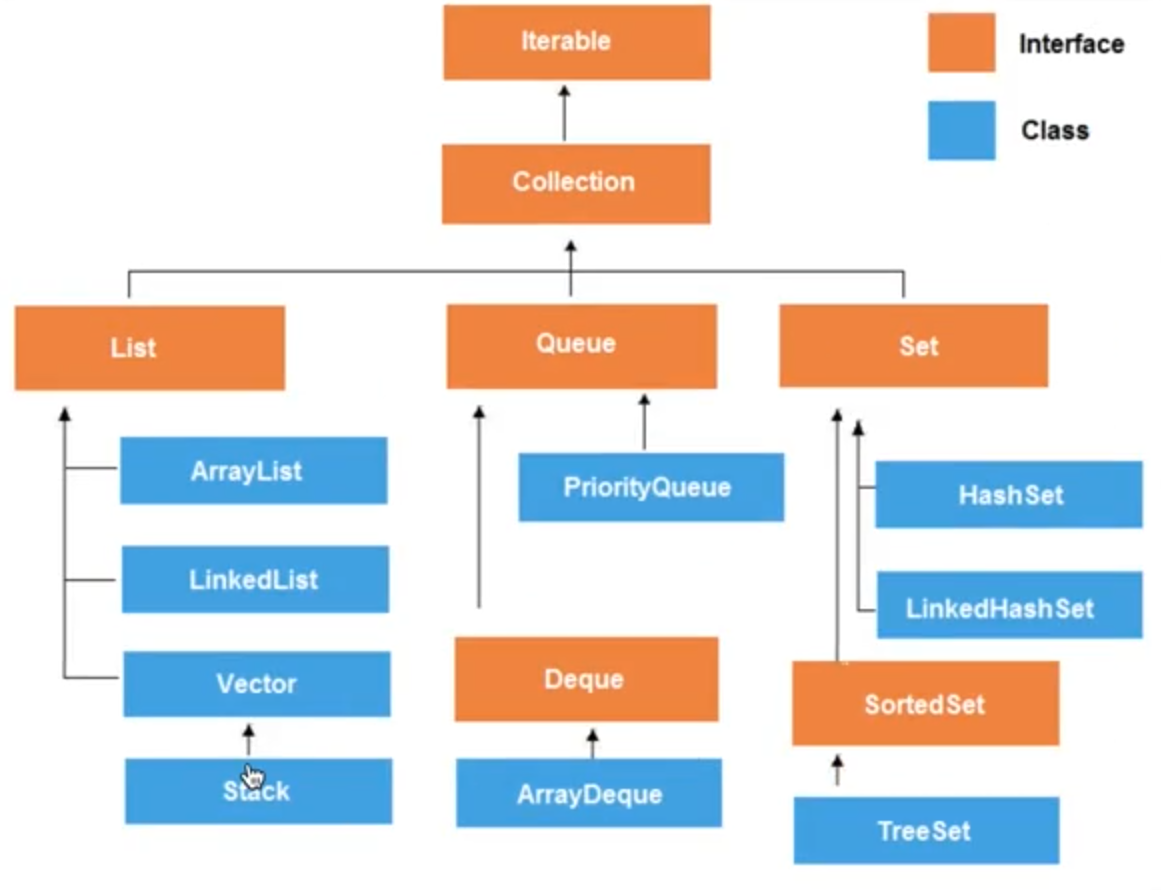
ArrayDeque = Array + Ring Buffer

Он настолько лёгкий и быстрый, что почти всегда быстрее LinkedList.

**УРОК 15 – СТЕК STACK**

На протяжении всех уроков мы разбирались с важнейшими и наиболее часто используемыми классами в Java Collections Framework и на этом уроке мы закончим разбираться с этими классами. И разберемся с последним классом, который входит в группу важных и наиболее часто используемых – классом Stack

На нашей привычной диаграмме класс Stack не изображен, поэтому возьмем другую диаграмму. Мы видим, что он у нас реализует интерфейс List и наследуется от класса Vector.



Класс Vector мы не разобрали и разбирать не будем, потому, что этот класс устаревший. В первых версиях он не существовал и он служил в качестве такого ArrayLista. По функционалу он был схож с ArrayListом, но с появлением последнего он стал deprecated (устаревший) и теперь этот класс не используется. Но класс Stack все еще наследуется от класса Vector.

Теперь рассмотрим сам класс Stack. Перейдем в среду разработки и создадим новый объект класса Stack.

public class Test {  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
 }  
}

Класс Stack тоже параметризован, как и все остальные классы в Java коллекциях.

Класс Stack работает по принципу LIFO (Last In First Out). Пример: когда в лифте одна дверь, лифт является Стеком. Первые зашедшие в него выйдут последними, а последние зашедшие – первыми.

Теперь разберемся с методами в классе Stack

**1. PUSH**

stack.push(1);

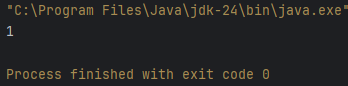
Push в переводе с английского – заталкивать. Так и здесь, метод push нужен для добавления новых элементов в стек – аналог add.

**2. POP**

stack.pop();

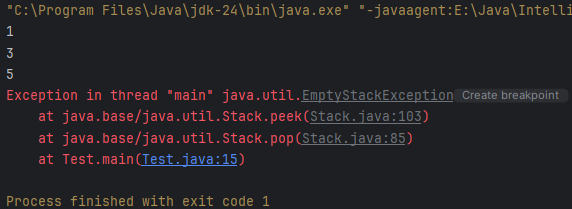
Pop достает последний добавленный в стек элемент.

System.*out*.println(stack.pop());



Если мы попытаемся достать из стека элемент, которого нет, то получим исключение:

System.*out*.println(stack.pop());  
System.*out*.println(stack.pop());  
System.*out*.println(stack.pop());  
System.*out*.println(stack.pop());



**3. PEEK**

Этот метод такой же как и Queue. Он смотрит на последний элемент в стеке, но не убирает его из стека

stack.peek();

**4. EMPTY**

Возвращает true, если стек пустой и false, если в нем что-то есть

stack.empty();

С помощью метода empty выведем все элементы стека на экран. (Получим мы их в обратном порядке, т.к. стек возвращает элементы с конца)

while(!stack.empty()){  
 System.*out*.println(stack.pop());  
}

Пока стек не пуст -> достаем элемент.

**УРОК 16 –** ИНТЕРФЕЙС ITERABLE

Это последний урок в цикле по Java Collections Framework и на этом уроке мы поговорим про интерфейс Iterable и про метод Itrader и зачем нужны этот интерфейс и метод.

Если мы вспомним предыдущие уроки, то мы вспомним, что мы реализовывали методы hashCode и equals для того, что проверять объекты на равенство в коллекциях, реализовывали метод compareTo, что установить естественный порядок и метод compare, чтобы сортировать элементы в коллекциях. И мы поняли, что в Java коллекциях (ArrayList, LinkedList и т.д. в Java Collections Framework) нет магии. Весь функционал, который есть в классах, он где-то реализован.

Поэтому, чтобы проверять объекты на равенство, нужно было реализовать методы. Чтобы сортировать объекты – также нужно было реализовать методы.

В этом уроке мы поймем, как работает итерация по нашим коллекциям.

Создадим новый ArrayList, добавим в него три элемента и выведем их на экран через цикл for each.

List<Integer> list = new ArrayList<>();  
list.add(1);  
list.add(2);  
list.add(3);  
  
for (int x : list){  
 System.*out*.println(x);  
}

В цикле for each мы проходимся по всем элементам list и мы видим, что у нас элементы поочередно выводятся на экран.

Почему это возможно? Можно подумать, что класс ArrayList такой магический и Java знает, что это коллекция и поэтому разрешает нам проходиться по всем элементам lista.

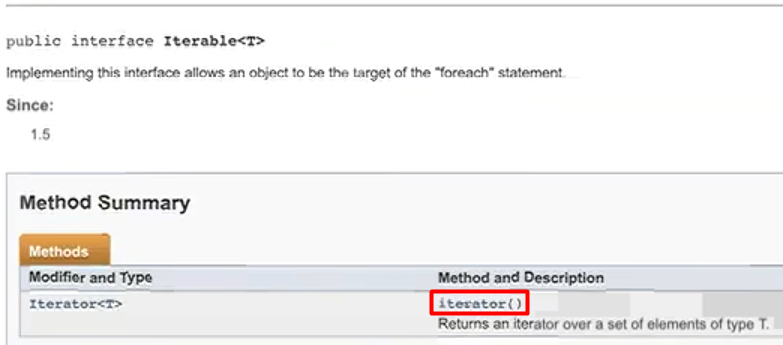
Но на самом деле это не так. Класс ArrayList это такой же класс, как и все другие классы в Java. И есть что-то, что говорит джаве о том, что этот класс не просто класс, но еще и сущность, по которой мы можем проходиться с помощью того же цикла for each. Эта сущность является коллекцией и хранит в себе какие-то элементы и по этим элементам мы хотим проходиться с помощью for each.

И вот то, что дает джаве понять, что по этой сущности можно проходиться с помощью for each, и называется интерфейсом Iterable.

Iterable в перевод с англ означает “то, по чему можно проходиться, то, что можно перебрать”.

Посмотрим на этот интерфейс в спецификации Java.

Мы увидим, что интерфейс Iterable имеет только один метод iterator()

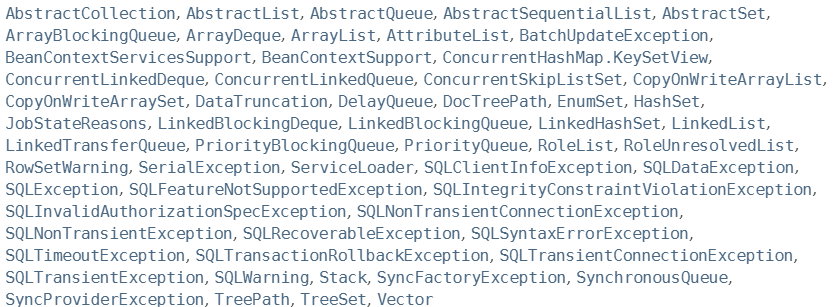


И этот метод возвращает итератор на сете элементов типа Т



Если перевести на язык понятный человеку, то: возвращает объект класса Iterator, который позволяет проходиться по элементам коллекции, которые переданы в качестве параметра T .

И мы видим, что очень много классов реализуют интерфейс Iterable:



И если нужно, чтобы по нашему собственному классу можно было проходиться циклом foreach, перебирая его элементы, то нам нужно в этом классе реализовать интерфейс Iterable. Но это не очень частая практика, поэтому мы разеберем принцип работы итератора в классах Java Collection Framework, конкретно в ArrayListe.

Цикл foreach появился в java 5 и foreach является своего рода абстракцией, до этого было оформление ближе к реализации.

До java 5 мы работали с итераторами напрямую. И для того, чтобы пройтись по всем элементам нашей коллекции, мы должны были делать следующее:

* Мы получали доступ к объекту интерфейс Iterator, который лежит в классе ArrayList, который в себе хранит эти элементы. Iterator интерфейс параметризованный и параметр устанавливался в соответствии с типом элементов, которые будут перебираться.

Iterable<T> — это интерфейс, который говорит:

«Этот объект можно перебирать в цикле foreach».

У него только один метод:



То есть если класс реализует Iterable, он должен уметь вернуть итератор.