Лабораторная работа № 2

Изучение коллекций Java и особенностей их практического применения

Подготовка к работе:

1. Повторить тему «Коллекции Java». Изучить особенности использования коллекций **ArrayList**, **LinkedList**, **HashSet**, **TreeSet** и их основные методы;
2. Повторить синтаксис использования цикла **foreach** применительно к коллекциям;
3. Изучить интерфейсы **Comparable** и **Comparator**, которые представляют функциональность для сравнения объектов одного и того же класса.

Выполнение работы:

1. В среде Intellij IDEA создать проект Java, в котором реализовать ввод с консоли 10-ти произвольных строк и запись их в коллекцию **ArrayList<String>**, перестановку последней строки в начало списка (повторить такую перестановку 15 раз), ввод с консоли ещё 5-ти произвольных строк и замена ими последних 5-ти строк в списке, вывод содержимого коллекции на экран при помощи цикла **foreach**.

Основная информация об интерфейсе **List** и его реализациях **ArrayList** и **LinkedList**.

List.

Интерфейс **List** определяет коллекцию, в которой элементы размещаются в индексированных ячейках. У коллекции **List**, в отличие от массивов, автоматически изменяемый размер. Если места для добавления нового элемента не достаточно, то размер коллекции увеличивается в 1.5 раза (по умолчанию).

Целесообразно для повышения универсальности кода конкретные реализации **List** хранить в переменной типа **List**. Всё равно у **ArrayList** и **LinkedList** нет ни одного публичного метода, которого не было бы у **List**.

Основные методы:

.size();

.isEmpty();

.add(<T> t); добавить в конец. Возвращает всегда **true**;

.remove(<T> t); удаление объекта, возвращает **true**, если такой объект существовал и теперь удалён;

.contains(<T> t); есть ли такой элемент? возвращает **boolean**;

.clear();

.iterator();

.add(int index, <T> t); добавить в любое место по индексу;

.get(int index);

.set(int index, <T> t); замена элемента по индексу на новое значение, возвращает элемент, который был заменён;

.remove(int index); удаление по индексу, возвращает удалённый объект.

ArrayList.

Фактически это массив со встроенным **System.arraycopy()**.

Хранилищем значений является приватное поле **elementData** с типом значений **Object[]**. При использовании конструктора по умолчанию создаётся лист размером 10 (приватное финальное статик поле **DefaultCapacity** = 10), а именно массив из десяти элементов с типом, указанным в дженерике.

В классе имеется приватное поле **size**, которое рассчитывается как **elementData.length**. Если места для добавления нового элемента не достаточно, то вызывается нативный метод **System.arraycopy()**, копирующий содержимое массива в новую область памяти, а размер массива увеличивается в 1.5 раза.

При удалении элементов из списка текущее значение размера списка не уменьшается. Поэтому для экономии памяти иногда желательно использовать метод .**trimToSize()**.

Для создания **ArrayList** иногда удобно использовать возможность статической инициализации (по аналогии с созданием массивов), обеспечиваемую методом .**asList()** утилитарного класса **Arrays**. Например:

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);

LinkedList.

Имплементирует не только интерфейс **List**, но и интерфейс **Deque**. От интерфейса **Deque** получает методы:

.getFirst();

.getLast();

.removeFirst();

.removeLast();

.addFirst();

.addLast();

.remove(); аналогичен .removeFirst();

.poll(); извлекает и удаляет первый элемент списка;

.peek(); извлекает, но не удаляет первый элемент списка;

.push(E); вставляет элемент в начало списка

.pollLast(); извлекает и удаляет последний элемент списка;

.peekLast(); извлекает, но не удаляет последний элемент списка.

В классе имеется поле **first** с типом **Node<E>** – указатель на первый узел в двухсвязном списке. И поле **last** с типом **Node<E>** – указатель на последний узел в двухсвязном списке.

Node<E> – это приватный класс с полями **E item**, **Node<E> next**, **Node<E> prev**.

Следовательно каждый узел хранит значение своего элемента **item**, а также ссылки на следующий и предыдущий узел.

Методы .**hasNext()** и .**next()**, а также .**hasPrevious()** и .**previous()**, вызываемые из объекта **Iterator**, позволяют пройти по цепочке от начала списка в конец или обратно.

2. В рамках проекта в среде Intellij IDEA сравнить быстродействие коллекций **ArrayList** и **LinkedList** на 100000 последовательных операций:
   - вставки нового значения в начало списка.
   - чтения каждого из 100000 элементов списка.
   - записи нового значения в каждый из 100000 элементов списка.
   - удаления первого элемента списка.

Любая упорядоченная коллекция работает медленнее, чем неупорядоченная. Поэтому любой **List** работает медленнее, чем **Set**.

Доступ к спискам можно разделить на 2 операции – поиск элемента и его модификация. Для **ArrayList** поиск по индексу выполняется моментально, но модификация происходит медленно (когда требуется **System.arraycopy()**). Для **LinkedList**, наоборот, модификация выполняется мгновенно, но поиск элемента (не важно, поиск по индексу или поиск объекта через .**contains(<T> t)** или для .**remove(<T> t)**) долго, если только элемент не располагается в начале или в конце списка (так как к нему надо пройти по цепочке от одного из концов).

3. В рамках проекта в среде Intellij IDEA:
   - создать два одинаковых по содержимому списка **ArrayList<Integer>**. Сравнить объекты списков при помощи метода .**equals()**.

- создать список **ArrayList<User>** для хранения объектов пользовательского класса **User**. Создать класс **User** с полями name (String), nickname (String), age (int). Обеспечить корректную работу методов .**contains(User user)** и .**remove(User user)** при проверки на наличие в списке и удалении одного из объектов списка, а также корректное сравнение двух списков с одинаковым содержимым.

У списков **ArrayList** и **LinkedList** переопределён метод .**equals()**, что позволяет сравнивать между собой списки по содержимому (поэлементно), а не по ссылкам на объекты списков. Но для того, чтобы появилась такая возможность необходимо также переопределить метод .**equals()** у объектов, которые помещаются в коллекцию. Чтобы сравнивались не ссылки на объекты, а содержимое объектов. У системных классов, таких как **Integer** и **String**, этот метод уже переопределён.

Целесообразно в классе объекта переопределять метод .**equals()** таким образом:

```
public boolean equals(Object obj){
if(obj == null)
return false;
if(this.getClass() != obj.getClass())
return false;
// далее приведение объекта к классу текущего объекта
Person that = (Person) obj;
// затем поэлементное сравнение объектов that и this и возвращение true при полном совпадении
}
```

Работа методов .**contains(<T> t)** и .**remove(<T> t)** зависит от того, переопределён ли метод .**equals()**. Если в аргументах этих методов через **new** создать объекты, которые запрашиваются у коллекции, то при корректном .**equals()** будет сравниваться содержимое этих объектов с содержимым объектов в коллекции, а не ссылки на эти объекты.

4. В рамках проекта в среде Intellij IDEA:
   - создать коллекцию **HashSet<String>** и поместить в неё фамилии студентов группы. Вывести содержимое коллекции на экран при помощи цикла **foreach**. Организовать удаление из коллекции фамилий, начинающихся на гласную букву. Вывести итоговое содержимое коллекции на экран.

- создать коллекцию **TreeSet<String>** и поместить в неё фамилии студентов группы. Вывести содержимое коллекции на экран при помощи цикла **foreach**. Организовать удаление из коллекции первых пяти фамилий (в алфавитном порядке). Вывести итоговое содержимое коллекции на экран.

Основная информация об интерфейсе **Set** и его реализациях **HashSet** и **TreeSet**.

Set.

Интерфейс **Set** определяет коллекцию в виде набора уникальных элементов, которые располагаются в коллекции в неочевидном порядке (вернее, конкретный порядок размещения определяется конкретной реализацией).

Основные методы:

.size();

.isEmpty();

.add(<T> t); добавить элемент. Возвращает **true**, если элемент добавлен. Возвращает **false**, если элемент не уникальный;

.contains(<T> t); есть ли такой элемент? возвращает **boolean**;

.remove(<T> t); удаление объекта, возвращает **boolean**;

.clear();

.iterator().

Реализации интерфейса **Set** не обязаны обеспечивать порядок элементов, но могут располагать элементы в некотором порядке. Например, в **HashSet** элементы располагаются в соответствии с номером в хэш-таблице. В **TreeSet** элементы располагаются в отсортированном порядке. В **LinkedHashSet** элементы располагаются в соответствии с порядком их добавления.

HashSet.

Работает намного быстрее списков. Для метода .**contains(<T> t)** требуется порядка O(1) операций вместо O(N) операций.

Для **HashSet** также необходимо переопределять метод .**equals()** в классах объектов, которые будут помещаться в **HashSet**, так чтобы сравнивалось содержимое объектов, а не их ссылки. Но так же обязательно надо переопределить метод .**hashCode()** так, чтобы он вычислял хэш-код объекта на основании содержимого его полей. Например так:

public int hashCode(){

return 31x + y; (если x и y – значения полей объекта, по которым идёт сравнение при помощи .**equals()**)

}

При этом **HashSet** будет работать максимально быстро. За счёт того, что в **HashSet** сравнение идёт сначала по хэш-кодам (это очень быстро, сравнение двух **int** это один такт процессора), а затем малое количество объектов с одинаковыми хэш-кодами сравниваются через .**equals()** (который работает медленней).

Принцип работы **HashSet**:

- Конструктор создаёт сет с размером 16 ячеек (по умолчанию). В каждой ячейке содержится **null**.

- При добавлении нового элемента в сет (например, при помощи метода .**add(<T> t)**) для него рассчитывается хэш-код при помощи метода .**hashCode()**. Хэш структуры работают максимально быстро когда хэш-коды всех элементов разные, но это трудно реализовать. Метод .**hashCode()** по умолчанию (в методе **Object**) возвращает **int**, но нигде не сказано, что хэш-коды элементов должны быть уникальны. Даже если переопределить .**hashCode()**, то всё равно сложно добиться абсолютной уникальности элементов. Алгоритм **31x + y**, используемый при переопределении .**hashCode()**, обеспечивает невыполнение свойства коммутативности (x+y=y+x). Это сделано для того, чтобы объекты с одинаковыми значениями элементов, но расположенными в разном порядке, давали разные хэш-коды.

- Каждый хэш-код преобразуется в диапазон номеров ячеек, соответствующий размеру **HashSet**. Например, если сет из 16 ячеек, то это диапазон 0-15. Для этого можно у каждого из хэш-кодов в двоичном виде взять младшие 4 разряда. Чтобы было удобно использовать такой подход размер сета всегда равен степени двойки.

- В ячейке **HashSet** с полученным номером создаётся ссылка на односвязный список. В первую ячейку созданного списка помещается добавляемый элемент. А в поле **next** ячейки, которое должно содержать ссылку на следующую ячейку, записывается **null**.

- Если добавляется элемент, у которого хэш-код другой, но младшие 4 разряда (в нашем примере) совпадают с номером ячейки для уже добавленного элемента, то новый элемент записывается в начало того же односвязного списка. А в его поле **next** записывается ссылка на второй (добавленный ранее) элемент. И так далее.

- Если добавляется объект, у которого хэш-код такой же, как у уже содержащегося в сете объекта, то для этих двух объектов вызывается метод .**equals()**. Если .**equals()** вернул **false**, то это разные объекты и новый элемент добавляется в начало соответствующего односвязного списка. Если .**equals()** вернул **true**, то такой новый элемент не добавляется. Поэтому так важно переопределять у объектов, добавляемых в сет, метод .**equals()**.

Чтобы сравнивалось содержимое объектов, а не их ссылки. Ведь в сет нельзя добавлять элементы с неуникальным содержимым.

- Для поиска элемента в **HashSet** сначала используется поиск по хэш-коду (преобразованному в номер ячейки), затем в односвязном списке для данной ячейки выполняется поиск по полному хэш-коду, а если элементов с одинаковым хэш-кодом несколько, то они сравниваются при помощи .**equals()**.

Поэтому **HashSet** работает быстрее, если у него больше ячеек, т.е. больше различных (преобразованных) хэш-кодов. Ведь сравнивать хэш-коды намного проще, чем выполнять .**equals()**.

Когда добавляется элемент, который уже не помещается в сет, то размер **HashSet** автоматически увеличивается вдвое. При этом пересчитываются все (преобразованные) хэш-коды всех элементов и элементы перераспределяются. Это очень затратная операция. Поэтому лучше зарание определить большой размер **HashSet**.

В одной ячейке **HashSet** может быть несколько элементов, а другая может быть пустая, но при таком подходе в каждой корзине примерно один элемент. Это максимизирует скорость работы сета.

TreeSet.

Работает намного быстрее списков. Для метода .**contains(<T> t)** требуется порядка O(log2(N)) операций.

TreeSet хранит элементы в отсортированном порядке. Но только примитивные типы или **String** можно сохранять в данной коллекции без подготовительных действий. Потому что классы-обёртки примитивных классов (которые можно использовать в дженериках коллекций) и класс **String** имплементируют интерфейс **Comparable**. **TreeSet** может хранить только упорядоченные объекты. Поэтому, если необходимо добавить в **TreeSet** произвольные объекты, то необходимо, чтобы класс объектов имплементировал интерфейс **Comparable**. Или создать отдельный класс, реализующий интерфейс **Comparator** для сравнения объектов, которые будут размещаться в **TreeSet**.

Принято использовать интерфейс **Comparable** для объектов, которые имеют некоторый естественный (интуитивно понятный) порядок. Для задания объектам произвольного порядка следования элементов (порядка сравнения) рекомендуется использовать интерфейс **Comparator**. При этом можно создать несколько отдельных классов, реализующих по разному интерфейс **Comparator**, передавая объект нужного класса в конструктор **TreeSet**.

Для класса объектов, которые помещаются в **TreeSet**, не требуется переопределять методы .**hashCode()** и .**equals()**. **TreeSet**

их не использует, он сравнивает объекты используя интерфейсы **Comparable** или **Comparator**. Если объекты при сравнении оказываются равны, то в сет будет добавлен только один такой объект (самый первый).

В коллекции **TreeSet** удобно хранить элементы, которые требуется также отсортировать, т.к. **TreeSet** отсортирует их автоматически.

Для всех рассмотренных типов коллекций можно посредством метода .**iterator()** получить объект **Iterator**. Данный объект, а также вызываемые из него методы .**hasNext()** и .**next()**, можно использовать для перебора значений, хранящихся в коллекции. Например:

```
HashSet<Integer> set = new HashSet< >();
Iterator<Integer> iterator = set.iterator();
    while (iterator.hasNext())
    {
       int number = iterator.next();
     System.out.println(number);
    }
```

Здесь метод .**hasNext()** возвращает **true**, если перебраны ещё не все элементы коллекции, а метод .**next()** переставляет курсор итератора на следующий элемент и возвращает его значение.

Кроме этого, итератор неявно используется при организации работы цикла **foreach**.

5. В рамках проекта в среде Intellij IDEA:
   - создать коллекцию **HashSet<Student>** и поместить в неё объекты класса **Student**, соответствующие участникам бригады. Класс **Student** должен содержать информацию об имени студента, фамилии студента, названии группы и номере бригады. Проверить корректность работы методов .**contains(Student student)** и .**remove(Student student)**.
   - переписать коллекцию студентов в **TreeSet<Student>**. При этом обеспечить сортировку объектов внутри коллекции при помощи интерфейса **Comparable** в соответствии с алфавитным порядком фамилий студентов. Вывести содержимое коллекции на экран при помощи цикла **foreach**.
   - создать коллекцию **TreeSet<Student>** и поместить в неё объекты класса **Student**, соответствующие участникам бригады. Обеспечить сортировку объектов внутри коллекции при помощи интерфейса **Comparator** в соответствии с количеством букв в фамилиях и

алфавитным порядком имён студентов. Вывести содержимое коллекции на экран при помощи цикла **foreach**.

Чтобы реализовать интерфейс **Comparable** надо реализовать в классе объекта метод .**compareTo(T o)**, который должен возвращать число типа **int**. В методе сравнивается текущий объект с другим, который передаётся в аргументе. Если текущий объект меньше того, который передан в аргументе, то возвращаемое число должно быть int < 0. Если объекты равны, то int = 0. Если текущий объект больше того, который передан в аргументе, то int > 0. Сравнение должно происходить по одному из признаков класса **T**. Например:

```
public class OOP1 {
    public static void main(String[] args) {

        Set<Person> set = new TreeSet<>();
        set.add(new Person("Anna", 20));
        set.add(new Person("Nastya", 25));
        System.out.println(set);
    }
}

class Person implements Comparable<Person>{
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(Person that) {
        if(this.age < that.getAge())
            return -1;
        else if this.age > that.getAge())
            return 1;
        else
            return 0;
    }
}
```

В интерфейсе **Comparator** объявлен метод с двумя аргументами .**compare(T o1, T o2)**. Поэтому данный метод, который будет сравнивать два объекта, надо реализовать в отдельном классе. И передать объект этого класса в конструктор **TreeSet** при создании сета. Это позволяет реализовать интерфейс **Comparator**, в том числе, при помощи анонимного класса.

```
public class OOP2 {
    public static void main(String[] args) {

        Set<Person> set = new TreeSet<>(new ComparePerson());
        set.add(new Person("Anna", 20));
        set.add(new Person("Nastya", 30));
        System.out.println(set);
    }
}

class Person{
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}

class ComparePerson implements Comparator<Person> {
    @Override
    public int compare(Person p0, Person p1) {
    return p0.getAge() - p1.getAge();
    }
}
```