Урок 3

Коллекции

Виды контейнеров в Java: List, Map, Set. Основные реализации и приемы использования, проход по элементам коллекции, сравнение и сортировка элементов коллекции.

Коллекции

Класс ArrayList

Класс ArrayList представляет собой динамический массив, размер которого может увеличиваться и уменьшаться по мере необходимости, в отличие от стандартных массивов, которые после создания имеют фиксированную длину. Списочные массивы создаются с некоторой начальной ёмкостью (capacity). Когда же первоначальной емкости оказывается недостаточно, коллекция автоматически расширяется путём создания в памяти списка в полтора раза большего предыдущего и копирования в него данных. Под ёмкостью подразумевается размер базового массива, используемого для хранения элементов данного вида коллекции. Ёмкость наращивается автоматически по мере ввода элементов в списочный массив.

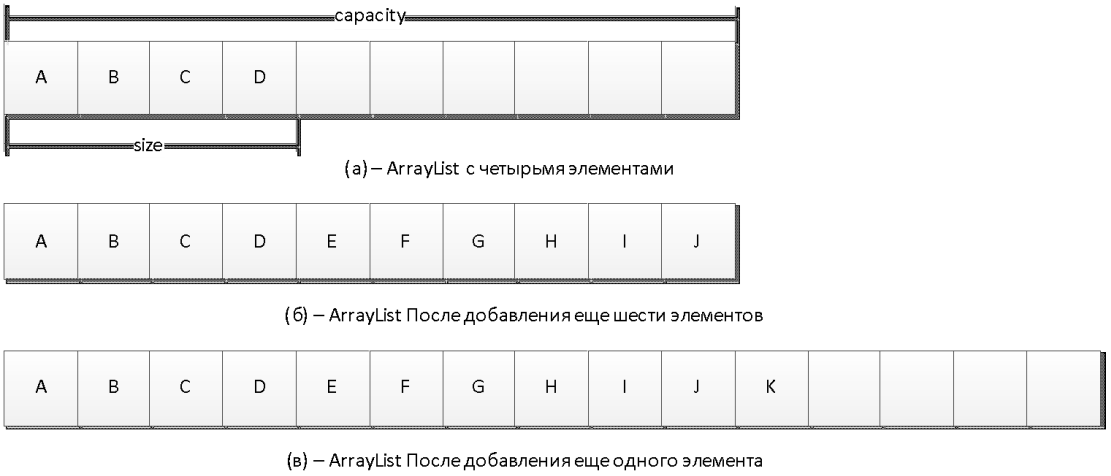

Рисунок 1 — Схема работы ArrayList

Класс ArrayList<E> является обобщенным, где E обозначает тип объектов, хранимых в списке. В классе ArrayList определены следующие конструкторы:

ArrayList()	Создает пустой ArrayList с начальной ёмкостью 10
ArrayList(Collection<? extends E> c)	Создает ArrayList, инициализируемый элементами из заданной коллекции c
ArrayList(int initialCapacity)	Создает ArrayList, имеющий указанную начальную емкость (initialCapacity)

Ниже приведена таблица с основными методами для работы с ArrayList.

Метод	Действие
boolean add(E e)	Добавить элемент в конец списка
void add(int index, E e)	Добавить элемент на позицию index
E get(int index)	Получить элемент списка с индексом index
void set(int index, E e)	Заменить элемент на позиции index
boolean remove(int index)	Удалить элемент списка с заданной позиции, вернуть — true, если объект был удален, false — в противном случае
boolean remove(E e)	Удалить заданный объект из списка, вернуть — true, если объект был удален, false — в противном случае
void trimToSize()	«Урезать» емкость списка до его размера
int size()	Получить размер списка
ensureCapacity(int capacity)	Увеличить емкости списка до значения capacity, только если текущая емкость меньше указанной
boolean contains(E e)	Проверить на присутствие указанного элемента в списке

В следующем примере демонстрируется простое применение класса ArrayList.

```
public static void main(String[] args) {
    ArrayList<String> al = new ArrayList<String>();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    al.add(1, "A0");
    System.out.println(al);
    al.remove("E");
    al.remove(2);
    System.out.println(al);
}
Результат:
[A, A0, B, C, D, E]
[A, A0, C, D]
```

Несмотря на то, что ёмкость объектов типа ArrayList наращивается автоматически, её можно увеличивать и вручную, вызывая метод ensureCapacity(), если заранее известно, что в коллекции предполагается сохранить намного больше элементов, чем она содержит в данный момент. Увеличив ёмкость списочного массива в самом начале его обработки, вы можете избежать дополнительного перераспределения памяти — дорогостоящей операции с точки зрения затрат времени.
С другой стороны, если требуется уменьшить размер базового массива, на основе которого строится объект типа ArrayList, до текущего количества хранящихся в действительности объектов, следует вызвать метод trimToSize().

Получение массива из коллекции

При обработке списочного массива типа ArrayList иногда требуется получить обычный массив, содержащий все элементы списка. Это можно сделать, вызвав метод toArray(). Имеется несколько причин, по которым возникает потребность преобразовать коллекцию в массив.

- Ускорение выполнения некоторых операций.
- Передача массива в качестве параметра методам, которые не перегружены для работы с коллекциями.
- Интеграция нового кода, основанного на коллекциях, с унаследованным кодом, который не

распознает коллекции.
Имеется два варианта метода toArray():

```
Object[] toArray();
<T> T[] toArray(T array[]);
```

В первой форме метод toArray() возвращает массив объектов типа Object, а во второй — массив элементов, относящихся к типу T. Обычно вторая форма данного метода удобнее, поскольку в ней возвращается надлежащий тип массива.

```
public static void main (String args[]) {
    ArrayList<Integer> a = new ArrayList<Integer>();
    a.add(1);
    a.add(2);
    a.add(3);
    Integer b[] = new Integer[a.size()];
    a.toArray(b);
}
```

Эта программа начинается с создания коллекции целых чисел. Затем вызывается метод toArray(), и получается массив элементов типа Integer. Коллекции могут содержать только ссылки, а не значения примитивных типов. Автоматическая упаковка позволяет передавать методу add() значения типа int, не прибегая к необходимости заключать их в оболочку типа Integer.

Класс LinkedList

LinkedList<E> предоставляет структуру данных связного списка, где E обозначает тип хранимых объектов. У класса LinkedList имеется два конструктора, аналогичных конструкторам ArrayList: LinkedList() и LinkedList(Collection<? extends E> c).
Каждый элемент в связном списке имеет ссылку на предыдущий элемент и на следующий. При поиске элемента по индексу необходимо будет обойти все элементы, стоящие на пути. При удалении же элемента не придётся смещать все элементы массива, а всего лишь переписать ссылки у двух элементов списка.

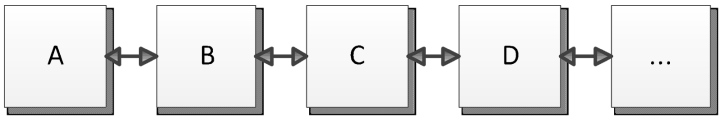

Рисунок 2 — LinkedList

Ниже приведена таблица с методами для работы с LinkedList.

Метод	Действие
addFirst(E e), offerFirst(E e)	Добавить элемент в начало списка
addLast(E e), offerLast(E e), add(E e)	Добавить элемент в конец списка
getFirst(), peekFirst()	Получить первый элемент списка
removeFirst(), pollFirst()	Получить первый элемент и удалить его из списка
getLast(), peekLast()	Получить последний элемент списка
removeLast(), pollLast()	Получить последний элемент и удалить его из списка

В следующем примере демонстрируется применение класса LinkedList:

```
public static void main(String args[]) {
    LinkedList<String> w = new LinkedList<String>();
    w.add("F");
    w.add("B");
    w.add("D");
    w.add("E");
    w.add("C");
    w.addLast("Z");
    w.addFirst("A");
    w.add(1, "A2");
    System.out.println("1. LinkedList w: " + w);
    w.remove("F");
    w.remove(2);
    System.out.println("2. LinkedList w: " + w);
    w.removeFirst();
    w.removeLast();
    System.out.println("3. LinkedList w: " + w);
    String val = w.get(2);
    w.set(2, val + " изменено");
    System.out.println("4. LinkedList w: " + w);
}
Результат:
1. LinkedList w: [A, A2, F, B, D, E, C, Z]
2. LinkedList w: [A, A2, D, E, C, Z]
3. LinkedList w: [A2, D, E, C]
4. LinkedList w: [A2, D, E изменено, C]
```

Обратите внимание, как третий элемент связного списка w изменяется с помощью методов get() и set(). Чтобы получить текущее значение элемента, методу get() передается индекс позиции, на которой расположен нужный элемент. А для того чтобы присвоить новое значение элементу на этой позиции, методу set() передаётся соответствующий индекс и новое значение.

Класс HashSet

Класс HashSet служит для создания коллекции, чтобы хранить элементы, в основе которой используется хеш-таблица. Для хранения данных в хеш-таблице применяется механизм хеширования, где содержимое ключа служит для определения однозначного значения, называемого хеш-кодом. Этот хеш-код служит в качестве индекса, по которому сохраняются данные, связанные с ключом. Преимущество хеширования заключается в том, что оно обеспечивает постоянство времени выполнения методов add(), contains(), remove() и size(). В классе HashSet определены следующие конструкторы:

HashSet()	Создает пустой HashSet с начальной ёмкостью 16 (по умолчанию)
HashSet(Collection<? extends E> c)	Создает HashSet, инициализируемый элементами из заданной коллекции c
HashSet(int initialCapacity)	Создает HashSet, имеющий указанную начальную емкость (initialCapacity)
HashSet(int initialCapacity, float loadFactor)	Создает HashSet, имеющий указанную начальную емкость (initialCapacity) и коэффициент заполнения (loadFactor) в пределах от 0.0 до 1.0

Коэффициент заполнения определяет, насколько заполненным должно быть хеш-множество, прежде чем будет изменена его емкость. В частности, когда количество элементов становится больше

ёмкости хеш-множества, умноженной на коэффициент заполнения, такое хеш-множество расширяется. В конструкторах, которые не принимают коэффициент заполнения в качестве параметра, выбирается значение этого коэффициента, равное 0.75.
В классе HashSet не определяется никаких дополнительных методов, помимо тех, что предоставляют его суперклассы и интерфейсы. Следует также иметь в виду, что класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств. Ниже приведён пример, демонстрирующий применение класса HashSet.

```
public static void main (String args[]) {
    HashSet<String> hs = new HashSet<String>();
    hs.add("Бета");
    hs.add("Альфа") ;
    hs.add("Эта");
    hs.add("Гамма") ;
    hs.add("Эпсилон");
    hs.add("Омега");
    System.out.println(hs);
}
Результат:
[Гамма, Эпсилон, Бета, Эта, Омега, Альфа]
```

Класс LinkedHashSet

Класс LinkedHashSet<E> расширяет класс HashSet, не добавляя никаких новых методов. Этот класс является обобщенным, где E обозначает тип объектов, которые будут храниться в хеш-множестве. У этого класса такие же конструкторы, как и у класса HashSet. В классе LinkedHashSet поддерживается связный список элементов хеш-множества в том порядке, в каком они введены в него. Это позволяет организовать итерацию с вводом элементов в определённом порядке.
Следовательно, когда перебор элементов хеш-множества типа LinkedHashSet производится с помощью итератора, элементы извлекаются из этого множества в том порядке, в каком они были введены. Именно в этом порядке они будут также возвращены методом toString(), вызываемым для объекта типа LinkedHashSet. Чтобы увидеть эффект от применения класса LinkedHashSet, попробуйте подставить его в исходный код предыдущего примера программы вместо класса HashSet. После этого выводимый программой результат будет выглядеть так, как показано ниже, отражая тот порядок, в каком элементы были введены в хеш-множество.

```
Результат:
[Бета, Альфа, Эта, Гамма, Эпсилон, Омега]
```

Класс TreeSet

Класс TreeSet создаёт коллекцию, где для хранения элементов применяет древовидная структура. Объекты сохраняются в отсортированном порядке по нарастающей. Время доступа и извлечения элементов достаточно мало, благодаря чему класс TreeSet оказывается отличным выбором для хранения больших объемов отсортированных данных, которые должны быть быстро найдены. Класс TreeSet<E> является обобщённым классом, где E обозначает тип объектов, которые будут храниться в древовидном множестве.
В классе TreeSet определены следующие конструкторы:

- TreeSet ().
- TreeSet (Collection<? extends E> c).
- TreeSet (Comparator<? super E> comparator).

- TreeSet (SortedSet<E> s).

В первой форме конструктора создаётся пустое древовидное множество, элементы которого будут отсортированы в естественном порядке по нарастающей. Во второй форме — древовидное множество, содержащее элементы заданной коллекции c. В третьей форме — пустое древовидное множество, элементы которого будут отсортированы заданным компаратором. И, наконец, в четвёртой форме создаётся древовидное множество, содержащее элементы заданного отсортированного множества s. В приведённом ниже примере программы демонстрируется применение класса TreeSet.

```
public static void main (String args[]) {
    TreeSet<String> ts = new TreeSet<String>();
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
    System.out.println(ts);
}
Результат:
[A, B, C, D, E, F]
```

Элементы такого множества автоматически располагаются в отсортированном порядке.

Класс HashMap

Класс HashMap<K, V> представляет собой хеш-таблицу для хранения отображения, благодаря чему обеспечивается постоянное время выполнения методов get() и put() даже в обращении к крупным отображениям. Класс HashMap<K, V> является обобщенным, где K обозначает тип ключей, а V — тип значений.

В классе определены следующие конструкторы:

HashMap()	Создает пустой HashMap с начальной емкостью 16 (по умолчанию)
HashMap(Map<? extends K, ? extends V> m)	Создает HashMap, инициализируемый элементами из заданного хеш-отображения m
HashMap(int initialCapacity)	Создает HashMap, имеющий указанную начальную емкость (initialCapacity)
HashMap(int initialCapacity, float loadFactor)	Создает HashMap, имеющий указанную начальную емкость (initialCapacity) и коэффициент заполнения (loadFactor) в пределах от 0.0 до 1.0

Назначение ёмкости и коэффициента заполнения такое же, как и в классе HashSet. По умолчанию, ёмкость составляет 16, а коэффициент заполнения — 0,75. Следует иметь в виду, что хеш-отображение не гарантирует порядок расположения своих элементов. Следовательно, порядок, в котором элементы вводятся в хеш-отображение, не обязательно соответствует тому порядку, в котором они извлекаются итератором. В следующем примере программы демонстрируется применение класса HashMap:

```
public static void main(String args[]) {
    HashMap<String, String> hm = new HashMap<>();
    hm.put("Russia", "Moscow");
    hm.put("France", "Paris");
    hm.put("Germany", "Berlin");
    hm.put("Norway", "Oslo");
    Set<Map.Entry<String, String>> set = hm.entrySet();
    for (Map.Entry<String, String> o : set) {
        System.out.print(o.getKey() + ": ");
        System.out.println(o.getValue());
    }
    hm.put("Germany", "Berlin2");
    System.out.println("New Germany Entry: " + hm.get("Germany"));
}
Результат:
Norway: Oslo
France: Paris
Germany: Berlin
Russia: Moscow
New Germany Entry: Berlin2
```

Выполнение данной программы начинается с создания хеш-отображения, в которое вводятся страны и столицы. Далее содержимое хеш-отображения выводится с помощью его представления в виде множества, получаемого из метода entrySet(). Ключи и значения выводятся в результате вызова методов getKey() и getValue(), определенных в интерфейсе Map.Entry. Обратите особое внимание на порядок изменения записи Germany/Berlin. Метод put() автоматически заменяет новым значением любое существовавшее ранее значение, связанное с указанным ключом. Таким образом, после обновления записи Germany/Berlin на Germany/Berlin2 хеш-отображение по-прежнему содержит только одну пару ключ/значение Germany/Berlin2.

Классы LinkedHashMap и TreeMap

Класс LinkedHashMap расширяет класс HashMap и является связным списком элементов, располагаемых в отображении в том порядке, в котором они в него добавлялись.
Класс TreeMap представляет собой отображение, размещаемое в древовидной структуре, хранит пары «ключ-значение» в отсортированном порядке (в порядке возрастания ключей) и обеспечивает их быстрое извлечение. В классе TreeMap определены следующие конструкторы:

TreeMap()	Создает пустой TreeMap с начальной емкостью 16 (по умолчанию)
TreeMap(Map<? extends K, ? extends V> m)	Создает TreeMap, инициализируемый элементами из заданного хеш-отображения m
TreeMap(SortedMap<K, ? extends V> sm)	Создает TreeMap, инициализируемый элементами из заданного хеш-отображения sm
TreeMap (Comparator<? super K> comparator)	Создает пустое древовидное отображение, которое будет отсортировано с помощью заданного компаратора типа Comparator

Домашнее задание

1 Создать массив с набором слов (10-20 слов, должны встречаться повторяющиеся). Найти и вывести список уникальных слов, из которых состоит массив (дубликаты не считаем).

Посчитать, сколько раз встречается каждое слово.

2 Написать простой класс Телефонный Справочник, который хранит в себе список фамилий и телефонных номеров. В этот телефонный справочник с помощью метода add() можно добавлять записи, а с помощью метода get() искать номер телефона по фамилии. Следует учесть, что под одной фамилией может быть несколько телефонов (в случае однофамильцев), тогда при запросе такой фамилии должны выводиться все телефоны.

Желательно как можно меньше добавлять своего, чего нет в задании (т.е. не надо в телефонную запись добавлять еще дополнительные поля (имя, отчество, адрес), делать взаимодействие с пользователем через консоль и т.д. Консоль желательно не использовать (в том числе Scanner), тестировать просто из метода main(), прописывая add() и get().

Дополнительные материалы

1 Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2 Брюс Эккель Философия Java // 4-е изд.: Пер. с англ. – СПб.: Питер, 2016. – 1 168 с.
3 Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4 Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.