Коллекции

Сложно придумать проект, в котором не требуется работа с коллекциями. Мы уже сталкивались с этим понятием в примере для отрисовки траектории робота в статье Визуализация робота. Также мы снова подняли этот вопрос в статье Список контактов – начало. Ну что же, пришло время с ними познакомиться. Коллекция в общем виде — это возможность собрать объекты в некоторую группу/множество и работать с этой группой. В общем-то самое главное уже сказано — коллекция предоставляет возможность совершать операции с группой объектов — это добавление, удаление, просмотр всех объектов в группе и прочие более специализированные операции. Еще раз — есть группа объектов, с которой надо совершать определенные операции и для этого нужен специальный класс. Вот этот класс по сути и есть коллекция. Я хотел бы выделить два важных момента:

1. Группа обычно имеет (я бы даже сказал — должна иметь) базовый набор функций — добавить, удалить, пройтись по всему списку, получить элемент. Но также возникает необходимость иметь дополнительные возможности, которые являются специфическими. Именно этим определяется разнообразие классов коллекций
2. Группа включает объекты в подавляющем большинстве случаев однотипных (одного класса). Хотя бывают исключения

Базовый функционал

Для работы с коллекциями разработчики Java создали специальный **Collection Framework**. Наиболее понравившееся мне руководство вот это: Trail: Collection (The Java Tutorials). Возможно я мог бы сказать — ну и читатйте сами его и наслаждайтесь, но все-таки я буду описывать свое понимание этого пакета. Но я не собираюсь рассказывать вам все-все-все. Это превратится в скучное перечисление функций, которое будет повторять официальную документацию, которую вам все равно придется читать во время работы. Не вижу никакого смысла этим заниматься. Но это не значит, что не надо читать другие источники — моя задача (такую я себе сам поставил) «наставить вас на путь истинный», по которому вы должны пойти сами.

Итак, вся система **Collection Framework** может быть разделена на три составляющих:

1. Набор базовых интерфейсов для нексольких типов коллекций
2. Набор классов для реализации базовых интерфейсов с разными «потребительскими» характеристиками
3. Набор алгоритмов для работы с коллекциями

Базовые интерфейсы

В официальной документации они все перечислены, но я не буду пока приводить его полностью, напишу пока самые важные (на мой взгляд конечно). Основная идея при рассмотрении этих интерфейсов должна быть такая — весьма умные люди разработали список методов, которые крайне важны для определенных типов коллекций — списков, множеств, очередей и прочая. Список имеет свои особенности, множество — свои, очередь — свои. Набор методов для списка и для множества будет различаться, т.к. эти типы коллекций (список и множество) имеют некоторые важные отличия. Рассматривайте их как специализированные инструменты — например, для закручивания шурупов нужен шуруповоерт, для бетонных стен — перфоратор, для сверления лунки — ледобур. Заметьте, что они все имеют «одну природу», но каждый имеет некоторую специализацию:

- **java.util.Collection** — основной интерфейс, который описывает базовые методы, которыми должна обладать любая коллекция. Т.е. если какой-то класс претендует на звание КОЛЛЕКЦИЯ — он должен реализовать те методы, которые описаны в этом интерфейсе. Проводя аналогию с нашим набором сверлильных инструментов — интерфейс **java.util.Collection** их общий родитель — у него есть возможнсть сверлить. Советую зайти на сайт с документацией и честно просмотреть все его методы. Возможно, что Java версии 8 (и выше) покажется вам сложноватой, поэтому для начала советую зайти на документацию по Java версии 7. java.util.Collection. Большая часть методов говорит сама за себя, так что почитайте.
- **java.util.List** — интерфейс для операций с коллекцией, которая является списком. Список обладает следующими важными признаками:
    1. Список может включать одинаковые элементы
    2. Элементы в списке хранятся в том порядке, в котором они туда помещались. Самопроизвольных перемещений элементов в нем не происходит — только с вашего ведома. Например, вы можете добавить элемент на какую-то позицию и тогда произойдет сдвиг других элементов.
    3. Можно получить доступ к любому элементу по его порядковому номеру/индексу внутри списка

  Т.е. если вам требуется, чтобы коллекция обладала такими свойствами — выбирайте класс, который реализует интерфейс **java.util.List**

- **java.util.Set** — интерфейс для хранения множества. В отличии от **java.util.List** этот интерфейс как раз не может иметь одинаковые элементы (смотрим методы **equals** и **hashCode** в статье Решения на основе классов) и порядок хранения элементов в множестве может меняться при добавлении/удалении/изменении элемента. Может возникнуть вопрос, зачем такиая коллекция нужна — это удобно в случае, когда вы создаете набор уникальных элементов из какой-то группы элементов
- **java.util.SortedSet** — это наследник интерфейса **java.util.Set** и его дополнительным функционалом является автоматическое выстраивание элементов внутри множества по порядку. Как этот порядок настаивается, мы поговорим позже.
- **java.util.Queue** — интерфейс предлагает работать с коллекцией как с очередью, т.е. коллекция имеет метод для добавления элементов в один конец и метод для получения элемента с другого конца — т.е. настоящая очередь по принципу FIFO — First In First Out — если первым пришел, то первым и уйдешь. Для широкого круга задач такая конструкция работы с коллекцией бывает достаточно удобной структурой.
- **java.util.Map** — очень удобная конструкция, которая хранит данные не в виде списка значений, а в виде пары ключ-значение. Это очень востребованная форма, в которой вы получаете доступ к значению в коллекции по ключу. Например, доступ к данным пользователя на сайте может быть осуществлен по логину (по email например). Самих данных может быть достаточно много, но для поиска можно использовать очень короткую строку-ключ.

И еще раз скажу самое важное — коллекция позволяет вам работать с группой объектов и специализация коллекции определяется требованиями к самим данным и к тем операциям, которые нужно использовать при работе с данными.

Простой пример использования коллекций

Прдлагаю посмотреть пример (демонстрацию) использования основных методов интерфейса **java.util.Collection**. Сначала просто напишу код примера и после этого прокомментирую его

```
1  package edu.javacourse.collection;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Iterator;
6
7
8  public class ExampleCollection
9  {
10     public static void main(String[] args) {
11         // Создаем коллекции для демонстрации
12         Collection col1 = createFirstCollection();
13         Collection col2 = createSecondCollection();
14
15         // Демонстрация прохода по коллекции
16         System.out.println("============= Проход по коллекции");
17         for(Object o : col1) {
18             System.out.println("Item:" + o);
19         }
```

```
20
21          System.out.println();
22          // Демонстрация прохода по коллекции через итератор
23          System.out.println("============= Проход по коллекции через итератор");
24          for (Iterator it = col1.iterator(); it.hasNext(); ) {
25              String s = (String)it.next();
26              System.out.println("Item:" + s);
27          }
28          System.out.println();
29
30          // Демонстрации групповых операций
31          System.out.println();
32          System.out.println("============= Групповые операции");
33          // Можно проверить сожержаться ли ВСЕ элементы col2 в col
34          if(col1.containsAll(col2)) {
35              System.out.println("Коллекция col содержит все от col2");
36          }
37
38          System.out.println("============= Добавление всех элементов в col1 из col2");
39          // Можно добавить элементы из col2 в col1
40          col1.addAll(col2);
41          for(Object o : col1) {
42              System.out.println("Item:" + o);
43          }
44
45          System.out.println("============= Удаление всех элементов col2, которые есть в col1");
46          // Можно удалить ВСЕ элементы col2, которые есть в col1
47          col1.removeAll(col2);
48          for(Object o : col1) {
49              System.out.println("Item:" + o);
50          }
51
52          // Пересоздаем коллекции для дпальнейшей демонстрации
53          col1 = createFirstCollection();
54          col2 = createSecondCollection();
55          System.out.println("============= Удаление элементов из col1, которых нет в col2");
56          col1.retainAll(col2);
57          for(Object o : col1) {
58              System.out.println("Item:" + o);
59          }
60          System.out.println("============= Очистка коллекции - не будет элементов");
61          col1.clear();
62          for(Object o : col1) {
63              System.out.println("Item:" + o);
64          }
65          System.out.println();
66
```

```java
        // Удаление элемента коллекции
        // Снова создаем коллекцию для демонстрации
        col1 = createFirstCollection();
        // Удаляем один элемент
        col1.remove("1");
        System.out.println("============= Удаляем элемент '1' - его не будет в списке");
        for(Object o : col1) {
            System.out.println("Item:" + o);
        }

        // Удаление коллекции через итератор
        // Снова создаем коллекцию для демонстрации
        col1 = createFirstCollection();
        System.out.println("============= Удаление через итератор");
        while(!col1.isEmpty()) {
            Iterator it = col1.iterator();
            Object o = it.next();
            System.out.println("Удаляем:" + o);
            // Удаляем элемент
            it.remove();
        }
    }

    // Первая коллекция для примера
    private static Collection createFirstCollection() {
        // Создать коллекцию на основе стандартного класса ArrayList
        Collection col = new ArrayList();
        // Добавление в коллекцию
        col.add("1");
        col.add("2");
        col.add("3");
        col.add("4");
        col.add("5");
        col.add("6");
        col.add("7");
        return col;
    }

    // Вторая коллекция для примера
    private static Collection createSecondCollection() {
        // Создать коллекцию на основе стандартного класса ArrayList
        Collection col2 = new ArrayList();
        col2.add("1");
        col2.add("2");
        col2.add("3");
        return col2;
    }
```

```
114 }
```

Если смотреть пример строчку за строчкой. то можно увидеть, какие именно функции используются и при запуске можно посмотреть результат выполнения этих функций. В принципе все достаточно просто — есть возможность добавлять в колллекцию элемент, есть возможность его удалять, есть возможность пройти по всему списку элементов и некоторые другие операции. Давайте смотреть маленькими кусочками и делать комментарии. Для начала рассмотрим два метода, где мы создаем коллекции.

```
1     // Первая коллекция для примера
2     private static Collection createFirstCollection() {
3         // Создать коллекцию на основе стандартного класса ArrayList
4         Collection col = new ArrayList();
5         // Добавление в коллекцию
6         col.add("1");
7         col.add("2");
8         col.add("3");
9         col.add("4");
10        col.add("5");
11        col.add("6");
12        col.add("7");
13        return col;
14    }
15
16    // Вторая коллекция для примера
17    private static Collection createSecondCollection() {
18        // Создать коллекцию на основе стандартного класса ArrayList
19        Collection col2 = new ArrayList();
20        col2.add("1");
21        col2.add("2");
22        col2.add("3");
23        return col2;
24    }
```

Как видите, все достаточно просто — мы берем нужный класс и создаем его экземпляр. Я для примера взял **java.util.ArrayList**. Т.к. этот класс реализует (имплементирует) интерфейс **java.util.Collection**, то у него есть все методы, которые в интерфейсе описаны. Добавление в коллекцию происходит очень просто — вызываем метод **add**. Вызывали — теперь ваш объект уже в коллекции. Добавляйте сколько угодно строк или другие объекты. У вас может возникнуть вопрос «Какие типы данных можно поместить в коллекцию ?». Вопрос резонный и я буду на него отвечать более подробно несколько позже. Пока можно сказать так — в коллекцию можно поместить объект любого класса, но нельзя туда поместить элементарный тип — int, char, long. Этот вопрос я тоже буду рассматривать более подробно чуть позже. Также может возникнуть еще один вопрос: «А почему мы создаем класс ArrayList, но присваиваем их ссылке на интерфейс Collection?». Вопрос тоже достаточно интересный и я попробую на него ответить. Дело в том, что при построении крупных систем очень выгодно интегрировать одтельные части через интерфейсы — контракты. Как один модуль реализует для другого этот контракт — это личное дело модуля-реализатора. Главное — выполнить контракт в полном объеме. И чем меньше надо

выполнять — тем проще. Мне в данном примере нужна коллекция и не больше. Значит есть смысл сознательно использовать только то, что надо и не «расстрачивать себя по мелочам». Вот такой вот практический посыл этой идеи.
А теперь давайте посмотрим наш код.

```
        // Создаем коллекции для демонстрации
        Collection col1 = createFirstCollection();
        Collection col2 = createSecondCollection();

        // Демонстрация прохода по коллекции
        System.out.println("============= Проход по коллекции");
        for(Object o : col1) {
            System.out.println("Item:" + o);
        }
```

В данном кусочке демонстрируется конструкция, которая позволяет обратиться к каждому элементу коллекции (очень похоже на массив — мы такое смотрели в разделе Массивы – знакомство. Обратите внимание — т.к. все классы наследуются от класса **Object**, то любой элемент в коллекции может рассматриваться как **Object**.
Здесь при каждом цикле мы помещаем в переменную **o** ссылку на следующий объект в коллекции. Т.е. у нас получается две ссылки — одна внутри самой коллекции, вторая — наша переменная .
Такая конструкция появилась только в Java 1.5. До этого для прохода по коллекции надо было использовать итератор:

```
        // Демонстрация прохода по коллекции через итератор
        System.out.println("============= Проход по коллекции через итератор");
        for (Iterator it = col1.iterator(); it.hasNext(); ) {
            String s = (String)it.next();
            System.out.println("Item:" + s);
        }
```

java.util.Iterator — это интерфейс, который позволяет перемещаться по списку элементов. При вызове метода **iterator()** вы получаете указатель на начало коллекции, но — ВНИМАНИЕ — не на первый элемент. Метод итератора **hasNext()** возвращает **true** в случае, если итератор может переместиться к следующему элементу (есть следующий за текущим), если получаем **false** — значит элементов больше нет.
Метод итератора **next()** перемещается на следующий элемент и возвращает его значение — объект типа . Еще раз — это объект типа **Object**. Обратите внимание — я здесь специально продемонстрировал (как я называю «жесткое») приведение типа — т.к. я знаю, что в коллекции находятся объекты типа **String**, то я преобразую ссылку на объект типа **Object** на ссылку на объект типа **String**.
Напомню, что с объектами мы работаем через ссылки и т.к. реальный объект в коллекции имеет тип **String**, то приведение не вызовет ошибок. Такое приведение позволит мне работать теперь с объектов как со строкой — там много всяких интересных методов есть, которых нет у **Object**.
Код для демонстрации групповых операций я предлагаю вам разобрать самим — в качестве самостоятельной работы. Там ничего особенного нет, так что дерзайте. Хочу выделить вот этот фрагмент:

```
        // Удаление элемента коллекции
        // Снова создаем коллекцию для демонстрации
        col1 = createFirstCollection();
        // Удаляем один элемент
        col1.remove("1");
        System.out.println("============= Удаляем элемент '1' - его не будет в списке");
        for(Object o : col1) {
            System.out.println("Item:" + o);
        }
```

Здесь мы удаляем элемент из коллекции. Но что весьма важно отметить — мы передаем ДРУГОЙ объект. Элемент, который мы передали методу **remove** и объект, который находится в коллекции — это ОДИНАКОВЫЕ, но РАЗНЫЕ объекты. По сути из коллекции удаляется объект, для которого метод **equals** возвращает **true** — смотрим раздел Решения на основе классов.
И на «закуску» сами разбираетесь в части, где удалаются все элементы через итератор

```
        // Удаление коллекции через итератор
        // Снова создаем коллекцию для демонстрации
        col1 = createFirstCollection();
        System.out.println("============= Удаление через итератор");
        while(!col1.isEmpty()) {
            Iterator it = col1.iterator();
            Object o = it.next();
            System.out.println("Удаляем:" + o);
            // Удаляем элемент
            it.remove();
        }
```

Как видите мы каждый раз после удаления проверяем не пуста ли наша коллекция (метод **isEmpty**) и если это не так, выставляем итератор в начало, переставляем его на первый элемент (т.к. коллекция не пустая, значит он точно есть) и удаляем.
Вот так вот на самом деле несложно получается работать с колекциями. Понятно, что дьявол кроется в деталях, но коллекции действительно сильно облегчают вам жизнь. Через некоторе время вы настолько к ним привыкните, что будете не понимать, как же без них раньше обходились.
Мы позже более подробно познакомимся с интерфейсами и классами, напишем небольшие примеры и я предложу вам самостоятельные задачки. Но сейчас мы поговорим о другой теме, которая имеет очень важную связь с коллекциями. В разделе Список контактов – начало мы использовали коллекцию для работы со списком контактов. Теперь вы даже можете новыми галазами посмотреть на наш класс **ContactSimpleDAO**. Там вы могли видеть вот такую конструкцию — описание функции **findContacts()** для возвращения списка контактов

```
public List<Contact> findContacts();
```

На интуитивном уровне возможно даже понятно, зачем нужна такая конструкция и что она делает, но сейчас самое время узнать о ней много интересного.

Что такое Generic

Давайте еще раз посмотрим на объявление нашего метода:

```
1 public List<Contact> findContacts();
```

Как видим, он возвращает список контактов (это класс, который реализует интерфейс **java.util.List**). Но что это за угловые скобочки, внутри которых находится слово **Contact** ? Давайте разбираться.

Старые песни на новый лад. Очередь объектов с Generic

Когда-то я написал статью Что нового в Java SE 5.0 и там упоминал о Generic. Так что кое-что можете посмотреть там тоже. Но и здесь мы будем изучать этот вопрос на примере, который мы разбирали раньше — Пример – очередь объектов.
Наша очередь, которую мы создали в этом примере, предоставляет возможность хранить список (произвольного размера) объектов любого типа. Но это достигается не самым приятным способом — мы вынуждены приводить полученные объекты к нужному нам типу из класса **Object**. С одной стороны — у нас очень гибкий инструмент. Мы туда можем положить и строки, и даты, и числа — все что угодно. Но на практике такое разнообразие типов в одном списке больше мешает, чем помогает. Идея создания класса, который бы мог работать с одной стороны с любым обхектом, а с другой стороны позволял четко определить с каким именно классом/типом он должен работать «на лету» достаточно продуктивна и поэтоу был создан механизм Generic.
Основную мысль я уже озвучил, так что повторюсь — Generic позволяет во-первых, определить класс, функциональность которого не зависит от типа объектов, с которыми он работает. И во-вторых, вы можете определить точный тип «на лету». Причем как только вы определяете тип, то все методы класса с Generic понимают только этот класс и никакой другой (дальше мы увидим, что это не совсем так, но для простоты я пока предлагаю понимать так). Как определяется Generic рассмотрим на самом простом примере, после которого перейдем к нашей очереди.
Итак, я хочу определить класс, у которого может быть поле произвольного класса и два метода — сеттер и геттер. Если не использовать Generic, то для универсальности мне пришлось бы написать что-то такое:

```
1 public class SimpleGeneric
2 {
3     private Object element;
4
5     public Object getElement() {
6         return element;
7     }
8
```

```
    public void setElement(Object element) {
        this.element = element;
    }
}
```

Использвать это класс для работы со строкой (**String**) пришлось бы как-то так

```
    SimpleGeneric sg = new SimpleGeneric();
    sg.setElement("12345");
    String s = (String) sg.getElement();
```

Самой важной здесь является третья строка с жестким приведением типа. Для одного случая это не самая большая проблема, но когда у вас сотни или даже тысячи разных классов, которые надо помещать в какие-то списки, очереди и т.д., то держать в голове, какие именно классы вы помещали в ту или иную очередь — это очень сложно и неудобно. Попробуем применить Generic.

```
public class SimpleGeneric<T>
{
    private T element;

    public T getElement() {
        return element;
    }

    public void setElement(T element) {
        this.element = element;
    }
}
```

Давайте внимательно посмотрим, что здесь изменилось. В первую очередь, в самом описании класса мы видим вот такую конструкцию

```
public class SimpleGeneric<T>
```

Буква «T» в угловых скобках говорит о том, что это — абстрактный тип (можно использовать не букву «T», а любое другое имя — например «SIMPLE» или «Test»). И этот абстрактный тип мы будем использовать в дальнейшем описании нашего generic-класса. Если я хочу вернуть объект из метода **getElement**, то я в описании указываю «T» вместо **Object**. Это означает, что наш класс в какой-то степени «полуфабрикат» — он заранее не знает, какой именно класс он будет использовать в этих методах. Но т.к. его алгоритм универсален (в данном случае мы делаем очень простые

функции, которые не зависят от класса), то мы как бы говорим нашему классу — «не парься, что именно за класс у тебя будет, в свое время узнаешь. И подставишь его вместо «T». Но чтобы ты понимал, в каких методах он встречается, мы тебе создаем подсказку в виде некоторого абстрактного типа «T» в угловых скобках». Именно угловые скобки говорят о том, что этот тип «для подстановки». Давайте посмотрим теперь как делать эту самую подстановку.

```
1  public class SimpleGenericTest
2  {
3      public static void main(String[] args) {
4          SimpleGeneric<String> sg1 = new SimpleGeneric<>();
5          sg1.setElement("12345");
6
7          SimpleGeneric<Integer> sg2 = new SimpleGeneric<>();
8          sg2.setElement(99);
9      }
10 }
```

Обратите внимание, что при объявлении переменной, мы в угловых скобках указываем конкретный класс. В первом случае это **String**, во втором — **Integer**. Т.е. при объявлении переменной мы уже точно определяем, с каким типом будет работать эта переменная. Теперь проверка правильности подстановки нужного типа проверяется уже на этапе компиляции. Например, если вы попробуете установить для переменной **sg2** строку — будет выдаваться ошибка. Попробуйте сделать вот так — и увидите.

```
1      SimpleGeneric<Integer> sg2 = new SimpleGeneric<>();
2      sg2.setElement("99");
```

До версии java 1.7 вы должны были дублировать содержиме угловых скобок и для правой части (точнее при вызове конструктора) — посмотрите пример.

```
1  public class SimpleGenericTest
2  {
3      public static void main(String[] args) {
4          // До версии Java 1.7 надо было указывать тип в конструкторе
5          SimpleGeneric<String> sg1 = new SimpleGeneric<String>();
6          sg1.setElement("12345");
7
8          SimpleGeneric<Integer> sg2 = new SimpleGeneric<Integer>();
9          sg2.setElement(99);
10     }
11 }
```

В работе с Generic-классами важно уловить самое важное — умение работать С ЛЮБЫМ КЛАССОМ. Ну или с некоторой группой классов, имеющих один и тот же интерфейс (мы это еще увидим). Коллекции в этом отношении — самые благодарные. Им в общем-то без разницы объект какого класса у них находится в работе — как грузовику совершенно не важно, коробки с какими надписями лежат у него в кузове.
Нашей очереди в общем не важно, какой именно тип будет использоваться. Но в этом случае объявление абстрактонго типа несколько сложнее, т.к. он используется сразу в двух классах — **ObjectQueue** и **QueueBox**. Давайте сначал рассмотрим вложенный класс **QueueBox**.

```
// Наш класс теперь Generic - тип хранимого объекта заранее неизвестен.
// Но нам это не важно - мы с ним по сути не работаем - не вызываем его методы
private class ObjectBox<T>
{
    // Поле для хранения объекта - теперь он не определен заранее
    private T object;
    // Поле для указания на следующий элемент в цепочке, у которого такой же тип.
    // Т.е. мы можем сказать. что использовать надо тот же тип T
    private ObjectBox<T> next;

    // Метод возвращает объект заранее неопределнного типа T
    public T getObject() {
        return object;
    }

    // Метод принимает на вход объект заранее неизвестного типа T
    public void setObject(T object) {
        this.object = object;
    }

    // Метод возращает указатель на следующий элемент в цепочке, у которого
    // тоже тип пока абстрактный, но такой же - T
    public ObjectBox<T> getNext() {
        return next;
    }

    // Метод принимает указатель на следующий элемент в цепочке, у которого
    // тоже тип пока абстрактный, но такой же - T
    public void setNext(ObjectBox<T> next) {
        this.next = next;
    }
}
```

Как видите тут описание сложнее. Наверно самое зубодробительное — это описание **private ObjectBox<T> next;**. Дело в том, что мы описываем переменную next и в этот момент мы должны указать, а какой тип будет использовать ObjectBox для этой переменной. Само собой мы не знаем. Но

мы знаем, что он будет таким же, что и у основного класса. Т.е. мы инициализируем next, но инициализируем опять же абстрактным класом. Поэтому такая хитрая конструкция.
На таком же принципе мы определяем и нашу очередь — класс **ObjectQueue**. Смотрим полный код класса

```
1  package edu.javacourse.queue;
2
3  public class ObjectQueue<T>
4  {
5      // Указатель на первый элемент
6      private ObjectBox<T> head = null;
7      // Указатель на последний элемент
8      private ObjectBox<T> tail = null;
9      // Поле для хранения размера очереди
10     private int size = 0;
11
12     public void push(T obj) {
13         // Сразу создаем вспомогательный объект и помещаем новый элемент в него
14         ObjectBox<T> ob = new ObjectBox<>();
15         ob.setObject(obj);
16         // Если очередь пустая - в ней еще нет элементов
17         if (head == null) {
18             // Теперь наша голова указывает на наш первый элемент
19             head = ob;
20         } else {
21             // Если это не первый элемент, то надо, чтобы последний элемент в очереди
22             // указывал на вновь прибывший элемент
23             tail.setNext(ob);
24         }
25         // И в любом случае нам надо наш "хвост" переместить на новый элемент
26         // Если это первый элемент, то "голова" и "хвост" будут указывать на один и тот же элемент
27         tail = ob;
28         // Увеличиваем размер нашей очереди
29         size++;
30     }
31
32     public T pull() {
33         // Если у нас нет элементов, то возвращаем null
34         if (size == 0) {
35             return null;
36         }
37         // Получаем наш объект из вспомогательного класса из "головы"
38         T obj = head.getObject();
39         // Перемещаем "голову" на следующий элемент
40         head = head.getNext();
41         // Если это был единственный элемент, то head станет равен null
```

```
        // и тогда tail (хвост) тоже дожен указать на null.
        if (head == null) {
            tail = null;
        }
        // Уменьшаем размер очереди
        size--;
        // Возвращаем значение
        return obj;
    }

    public T get(int index) {
        // Если нет элементов или индекс больше размера или индекс меньше 0
        if(size == 0 || index >= size || index < 0) {
            return null;
        }
        // Устанавлваем указатель, который будем перемещать на "голову"
        ObjectBox<T> current = head;
        // В этом случае позиция равну 0
        int pos = 0;
        // Пока позиция не достигла нужного индекса
        while(pos < index) {
            // Перемещаемся на следующий элемент
            current = current.getNext();
            // И увеличиваем позицию
            pos++;
        }
        // Мы дошли до нужной позиции и теперь можем вернуть элемент
        T obj = current.getObject();
        return obj;
    }

    public int size() {
        return size;
    }

    // Наш вспомогательный класс будет закрыт от посторонних глаз
    private class ObjectBox<T>
    {
        // Поле для хранения объекта
        private T object;
        // Поле для указания на следующий элемент в цепочке.
        // Если оно равно NULL - значит это последний элемент
        private ObjectBox<T> next;

        public T getObject() {
            return object;
        }
```

```
89
90          public void setObject(T object) {
91              this.object = object;
92          }
93
94          public ObjectBox getNext() {
95              return next;
96          }
97
98          public void setNext(ObjectBox<T> next) {
99              this.next = next;
100         }
101     }
102 }
```

Как видите, теперь мы везде используем абстрактный класс «T». Я вам советую внимательно разобраться в примере. Класс для проверки нашей очереди теперь выглядит вот так

```
1  package edu.javacourse.queue;
2
3  public class QueueTest
4  {
5      public static void main(String[] arg) {
6          ObjectQueue<String> queue = new ObjectQueue<>();
7
8          for(int i=0; i<10; i++) {
9              queue.push("Строка:" + i*100);
10         }
11
12         for(int i=0; i<queue.size(); i++) {
13             String s = queue.get(i);
14             System.out.println(s);
15         }
16
17         System.out.println("===============");
18
19         while(queue.size() > 0) {
20             String s = queue.pull();
21             System.out.println(s + " Размер:" + queue.size());
22         }
23     }
24 }
```

В первой же строке мы указываем, что наша очередь будет работать со строками. Теперь все методы, которые объявлены с «T» по сути заменят его на **String**. Наши методы становятся жестко типизированными (для переменной **queue**) — мы не сможем положить в нашу очередь **queue** числа или даты и при вызове метода **get** или **pull** методы будут сразу возвращать **String**. Что несомненно удобно. Причем если мы объявим переменную **queue2** и типизируем ее **Integer**, то для этой переменнй строки будут недоступны.
В качестве домашнего задания попробуйте переписать пример из раздела Визуализация робота с использованием нашей типизированной очереди. На этом я хочу закончить первое знакомство с generic-классами и перейти к коллекциям. Но мы еще вернемся к этой теме.

Классы из CollectionFramework

Итак, мы с вами поззнакомились с понятие generic и теперь пора познакомиться с некоторыми готовыми классами, которые уже написаны и включены в Java SE.
Когда вы приступаете к выбору класса для коллекции, то вам необходимо определить набор характеристик, которые являются важными в рамках тех задач, которые призвана решать эта коллекция. Мы уже прошлись по списку базовых интерфейсов и это первый шаг для выбора коллекции. Например, если у вас предполагается ситуация с одинаковыми элементами в коллекции или вам крайне важен порядок элементов в коллекции, да к тому же вам надо иметь возможность обратиться к элементу на определенной позиции, то практически однозначно вам нужны реализации интерфейса **List**. Если вам необходима уникальность, быстрый поиск элемента внутри коллекции и порядок не важен — смотрим в сторорну **Set**. Нужен отсортированный порядок уникальных элементов — смотрим **SortedSet**.
Если работа с коллекцией требует поиска элемента по ключу, то на первом месте идет **Map**.
После того, как вы определитесь с базовым интерфейсом, наступает время выбора конкретной реализации, что тоже требует знакомства с конкретными классами. Например вы остановили свой выбор на интерфейсе **List**. Если зайти на страницу с документацией List (Java Platform SE 8), то выбор конечно не огромный, но достаточный, чтобы задуматься. Например, что нам больше подойдет — **arrayList, LinkedList, Vector** ? Здесь начинается изучение особенностей реализации. Класс **ArrayList** — прекрасный выбор, если вам нужен быстрый доступ к элементу по индексу и вы заранее знаете (хотя бы приблизительно) сколько элементов будет в этой коллекции. Но если вам надо много добавлений и не требуется доступ к элементу, а нужно пробегать по всему списку, то этот класс невыгоден — он основан на массиве и как только вы достигаете определенного размера, то будет создаваться новый массив, в который будут копироваться все элементы. Это дорогое удовольствие. Зато класс **LinkedList** прекрасно подходит. Мы уже в этом убедились — ведь мы создавали по сути свою версию этого класса — класс **ObjectQueue**. Могут быть дополнительные требования — например потокобезопасность (мы будем говорить о потоках позже). В этом случае возможно подойдет что-то еще. Т.е. с одной стороны все выглядит достаточно несложно — надо просто выбрать подходящий класс, но как говорится «дьявол кроется в деталях». Я встречал ситуации, когда разработчики создавали свои версии коллекций, т.к. существующие им не подходили.

Алгоритмы для работы с коллекциями

Алгоритмы для работы с коллекциями реализуются (в основном) в классе **java.util.Collections**. Не перепутайте с классом, о котором мы уже говорили — **java.util.Collection**. Отличие минимальное — в конце стоит дополнительная буква **s**. Если честно, мне не нравится — слишком похожие имена легко путаются, что меня никогда не радовало. Но что сделано, то сделано.
Я надеюсь, что вы уже достаточно продвинулись в изучении java и тратить кучу слов на описание методов мне бы не хотелось. Поэтому сразу предлагаю посмотреть пример — в нем последовательно приведены вызовы с некоторым набором функций класса **java.utilCollections**. просто посмотрите код, потом запустите и посмотрите на результаты. Весь пример посвящен нескольким операциям, результат которых демонстрируется выводом коллекции

```java
package edu.javacourse.collection;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * Демонстрация различных операций над списком
 */
public class Main {

    public static void main(String[] args) {

        List<MyClass> list = new ArrayList<MyClass>();

        list.add(new MyClass("Василий"));
        list.add(new MyClass("Павел"));
        list.add(new MyClass("Андрей"));
        list.add(new MyClass("Андрей"));
        list.add(new MyClass("Петр"));
        list.add(new MyClass("Анжелика"));

        printCollection("Оригинал", list);

        // Смешивание
        Collections.shuffle(list);
        printCollection("Смешивание", list);

        // Обратный порядок
        Collections.reverse(list);
        printCollection("Обратный порядок", list);

        // "Проворачивание" на определенное количество
        Collections.rotate(list, 2); // Число может быть отрицательным - тогда порядок будет обратный
        printCollection("Проворачивание", list);

        // Обмен элементов
        Collections.swap(list, 0, list.size()-1);
        printCollection("Обмен элементов", list);

        // Замена
        Collections.replaceAll(list, new MyClass("Андрей"), new MyClass("Алексей"));
        printCollection("Замена", list);

        // Копирование - здесь обязательно надо иметь нужные размеры
```

```
        List<MyClass> list2 = new ArrayList<MyClass>(list.size());
        // Поэтому заполняем список. Хоть чем-нибудь.
        for(MyClass mc : list) {
            list2.add(null);
        }
        // Компируем из правого аргумента в левый
        Collections.copy(list2, list);
        printCollection("Копирование", list2);

        // Полная замена
        Collections.fill(list2, new MyClass("Антон"));
        printCollection("Полная замена", list2);

    }

    private static void printCollection(String title, List<MyClass> list) {
        System.out.println(title);
        for(MyClass mc : list) {
            System.out.println("Item:" + mc);
        }
        System.out.println();
    }

}
```

И наш «подопытный» класс, в котором надо обатить внимание на мметоды **equals** и **hashcode** —

при замене нам нужно находить равные объекты.

```
package edu.javacourse.collection;

/**
 * пример произвольного класса
 */
public class MyClass {

    private String name;

    public MyClass(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
```

```
        return name;
    }

    // Без методов equals и hashCode не будет замены в списках
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final MyClass other = (MyClass) obj;
        if ((this.name == null) ? (other.name != null) : !this.name.equals(other.name)) {
            return false;
        }
        return true;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 29 * hash + (this.name != null ? this.name.hashCode() : 0);
        return hash;
    }
}
```

Сортировка

Я специально выделил этот пункт, хотя сортировка тоже входить в стандартные агоритмы. Но т.к. она требует дополнительных действий, то поговорим мы о ней отдельно.
Я выделю два пути, которые позволяют сортировать коллекции:

1. Использовать коллекцию с реализацией интерфейса **java.util.SortedSet** — например класс **java.util.TreeSet**
2. Использовать метод **sort** класса **java.util.Collections**

В первом случае (и частично во втором тоже) на класс, который вы храните в коллекции, накладывается требование — он должен реализовать интерфейс . При использовании **SortedSet** сортировка происходит сразу после добавления. Я предлагаю вам проверить это в качестве домашнего задания. В случае использования метода **sort** класса **java.util.Collections** вам потребуется создать класс, который реализует интерфейс **java.util.Comparator**. Задача этого класса очень простая — он должен сравнить два объекта. Предлагаю опять же рассмотреть все на примере. У нас будет 4 класса:

1. MyClass — самый обычный класс, который мы сможем отсортировать

2. MyClassCompare — класс, которые реализует интерфейс **Comparable**
3. MyClassComparator — реализация интерфейса **java.util.Comparator**
4. Main — класс для демонстрации

```java
package edu.javacourse.collection;

public class MyClass
{
    private String name;

    public MyClass(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

```java
package edu.javacourse.collection;

public class MyClassCompare implements Comparable<MyClassCompare>
{
    private String name;

    public MyClassCompare(String name) {
        this.name = name;
    }

    // этот метод как раз и сравнивает текущий объект с другим,
    // который передается в качестве аргумента
    public int compareTo(MyClassCompare o) {
        return name.compareTo(o.name);
    }

    @Override
    public String toString() {
        return name;
    }
}
```

```java
package edu.javacourse.collection;

import java.util.Comparator;

public class MyClassComparator implements Comparator
{
    public int compare(Object o1, Object o2) {
        return o1.toString().compareTo(o2.toString());
    }
}
```

```java
package edu.javacourse.collection;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main
{
    public static void main(String[] args) {
        System.out.println("Вариант сортировки через Comparator");
        List<MyClassCompare> listSort = new ArrayList<MyClassCompare>();
        listSort.add(new MyClassCompare("Василий"));
        listSort.add(new MyClassCompare("Павел"));
        listSort.add(new MyClassCompare("Андрей"));
        listSort.add(new MyClassCompare("Андрей"));
        listSort.add(new MyClassCompare("Петр"));
        listSort.add(new MyClassCompare("Анжелика"));
        // Сортировка через Comparable
        printCollection1("Без сортировки", listSort);
        Collections.sort(listSort);
        printCollection1("После сортировки", listSort);

        System.out.println("Вариант сортировки через Comparator");
        List<MyClass> list = new ArrayList<MyClass>();
        list.add(new MyClass("Василий"));
        list.add(new MyClass("Павел"));
        list.add(new MyClass("Андрей"));
        list.add(new MyClass("Андрей"));
        list.add(new MyClass("Петр"));
        list.add(new MyClass("Анжелика"));
        // Сортировка с классом Comparator
        printCollection2("Без сортировки", list);
```

```
        Collections.sort(list, new MyClassComparator());
        printCollection2("После сортировки", list);
    }

    private static void printCollection1(String title, List<MyClassCompare> list) {
        System.out.println(title);
        for (MyClassCompare mc : list) {
            System.out.println("Item:" + mc);
        }
        System.out.println();
    }

    private static void printCollection2(String title, List<MyClass> list) {
        System.out.println(title);
        for (MyClass mc : list) {
            System.out.println("Item:" + mc);
        }
        System.out.println();
    }
}
```

Я выделю следующие моменты:

- Класс **MyClassCompare** реализует метод **compareTo**. Он возвращает число. Если оно больше нуля, то объект больше того, который передан в аргументе, если равен нулю — объекты равны, если меньше — переданный объект больше. Т.к. стандартные классы (в том числе и **String**) реализуют интерфейс **Comparabe**, то я им и воспользовался. Также обратите внимание, что интерфейс является generic — т.е. мы можем заранее сказать, какие классы будут сравниваться.
- Класс **MyClassComparator** занимается сравнением двух объектов (отметим, что интерфейс **Comparator** тоже generic) с помощью метода **compare**, который работает по такому же принципу, что и метод **compareTo** у интерфейса **Comparabe**
- Методы **printCollection1** и **printCollection2** выглядят очень похоже, но т.к. коллекции используют разные классы, мы вынуждены их разделить. Наверно это не так удобно, как хотелось бы — мы еще посмотрим, как с этим бороться

Хотелось бы отметить важный момент — при использовании класса от интерфейса **Comparator** сортируемый класс не должен реализовывать интерфейс **Comparable**. Также использование нескольких компараторов может позволить вам делать разные сортировки одной и той же коллекции.

Домашнее задание

1. Попробуйте модифицировать класс **MyClassComparator** так, чтобы он мог сортировать не только в алфавитном порядке — от А до Я, но и в обратном — от Я до А
2. Возьмите класс **Contact** из нашего проекта список контактов — начало и напишите для него компаратор, который может сортировать его по любому из полей

3. Более сложный вариант — сделайте так, чтобы при сортировке контактов вы могли передать список полей, по которым хотите его отсортировать

Удачи.

И теперь нас ждет следующая статья: Коллекции — продолжение.

14 comments to *Коллекции — базовые принципы*

- 
  Декабрь 6, 2017 at 10:14
  agarty says:

  А не могли бы Вы описать тип «Ыекштп» приведенный в статье? =D =D

  Reply

  - 
    Декабрь 6, 2017 at 10:50
    admin says:

    Спасибо — исправил.

    Reply

- 
  Декабрь 8, 2017 at 10:31
  agarty says:

  Интересно, а не вкралась ли ошибка в методе compareTo() класса MyClassCompare? в каменте написано, что текущий с другим, но в методе он сам с собой сравнивается. после смены на return name.compareTo(o.toString()); результат не изменился…

  Reply

Декабрь 8, 2017 at 10:43
agarty says:

сорян, изменился. я не туда смотрел. если name.compareTo(name) — результат будет без изменений, что логично. если же name.compareTo(o.toString()) — то происходит сравнение

Reply

Январь 11, 2018 at 21:55
Basilisk says:

в классе «MyClassCompare» в методе «compareTo()» опечатка в строке, где происходит возврат и сравнение: упущен аргумент «o.name», указан просто «name»;
{code}
public int compareTo(MyClassCompare o) {
return name.compareTo(o.name);
}
{/code}
P.S. спасибо за отличный курс

Reply

Январь 12, 2018 at 15:08
admin says:

Спасибо — исправил.

Reply

Май 8, 2018 at 18:26
Никита says:

Спасибо огромное за этот прекрасный курс!
Если я могу высказать пожелание, то хотелось бы немного больше домашних заданий — таких как к этому уроку. Иначе темы сменяются очень быстро, и материал не успевает усваиваться. И когда смотришь на страшные названия в последних темах — становится тревожно =)

Даже судя по количеству комментариев, до этой части курса доходят не все :P

Маленький вопрос по этой теме:
При добавлении элементов с помощью SortedSet, элементы с одинаковым именем не добавляются. Добавляется только первый из таких элементов. Это особенность коллекции такая? Есть ли способ быстро это исправить или проще использовать List?

Reply

Май 8, 2018 at 21:03
admin says:

Домашние задания — это отдельная тема, над которой надо работать отдельно.
Что касается Set — это такая коллекция. Она содержит только уникальные элементы (которые не равны). Если хотите использовать одинаковые элементы — берите List. Набивайте его данными и потом сортируйте. Вот тут кстати можно посмотреть. какая реализация будет быстрее работать при таком построении алгоритма.

Reply

Июнь 10, 2018 at 18:36
Денис says:

Спасибо за курс.
Вопрос по второму заданию

Научился сортировать коллекцию контактов, но только по айди(который стоит первым).
Не могу понять как сортировать по остальным значениям, подскажите, плиз

Reply

- 
  Июнь 12, 2018 at 17:42
  admin says:

  Если надо воспользоваться методом Collections.sort, то можно использовать реализацию интерфейса Comparator — ч статье он упоминается. Надо просто написать еще один компаратор.

  Reply

- 
  Август 6, 2018 at 17:35
  Сергей says:

  Добрый день. У меня вопрос касательно удаления элемента из коллекции:
  // Удаляем один элемент
  col1.remove(«1»);

  По факту получается что удаляется только первый попавшийся такой элемент ??
  Если я для примера в коллекцию добавлю еще несколько элементов «1», то удаляется только самый первый из них, а остальные остаются в коллекции…

  Reply

  - 
    Август 10, 2018 at 04:13
    admin says:

    Судя по всему, Вы используете реализацию интерфейса List. Но он так и работает — удаляет только один элемент, который подходит.

    Reply

Август 7, 2018 at 08:44
Сергей says:

Доброе утро….вчера добавлял комментарий, а его нет до сих пор, при попытке создать такой же, пишет что «попытка создания дубликата»…

Reply

Август 10, 2018 at 04:13
admin says:

Я сейчас не имею возможности часто администрировать комментарии. А они премодерируются во избежании спама.

Reply

Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

9 + [ ] = 16

Add comment