ЛАБОРАТОРНАЯ РАБОТА №9. КОЛЛЕКЦИИ

Цель работы. Получить практические навыки работы с коллекциями в Java.

9.1. Коллекции объектов Java

Коллекции объектов – это очень мощный и исключительно полезный механизм. Простейшей коллекцией является массив. Но массив имеет ряд недостатков. Один из самых существенных – размер массива фиксируется до начала его использования. Т.е. необходимо заранее знать или подсчитать, сколько потребуется элементов коллекции до начала работы с ней. Зачастую это неудобно, а в некоторых случаях – невозможно.

Поэтому все современные базовые библиотеки различных языков программирования имеют тот или иной вариант поддержки коллекций объектов. Коллекции обладают одним важным свойством – их размер не ограничен. Выделение необходимых для коллекции ресурсов спрятано внутри соответствующего класса. Работа с коллекциями облегчает и упрощает разработку приложений. Отсутствие же подобного механизма в составе средств разработки вызывает серьезные проблемы, которые приводят либо к различным ограничениям в реализации либо к самостоятельной разработке адекватных средств для хранения и обработки массивов информации заранее неопределенного размера.

В Java средства работы с коллекциями весьма развиты и имеют много полезных особенностей.

Одним из широко используемых **классов коллекций является ArrayList.** Пример использования этой коллекции приводится ниже (Листинг 8.1).

Листинг 8.1

```
import java.util.*;
import java.io.*;
public class ArrayListTest {
    ArrayList lst = new ArrayList();
    Random generator = new Random();
    void addRandom() {
        lst.add(new Integer(generator.nextInt()));
    }
    public String toString() {
        return lst.toString();
    }
    public static void main(String args[]) {
       ArrayListTest tst = new ArrayListTest();
       for(int i = 0; i < 100; i++ )
            tst.addRandom();
        System.out.println("Сто случайных чисел: "+tst.toString());
    }
}
```

Рассмотрим данный пример подробнее. Здесь, кроме **класса ArrayList**, использованы еще ряд классов библиотеки Java.

– **Random – класс из java.util.** Расширяет возможности класса **Math** по генерации случайных чисел.

– **Integer** – так называемый **wrapper-класс (класс-обертка)** для целых (int). Он использован потому, что в коллекцию нельзя занести данные элементарных типов, а только объекты классов.

Класс ArrayListTest имеет два поля – **поле lst класса ArrayList** и **поле generator класса Random**, используемое для генерации случайных чисел.

Метод addRandom() генерирует и заносит в коллекцию очередное случайное число. **Метод toString()** просто обращается к методу **toString()** класса **ArrayList**, который обеспечивает формирование представления списка в виде строки.

Метод main(...) создает объект класса **ArrayListTest** и организует цикл заполняющий коллекция 100 случайными числами. После этого он печатает результат.

Этот пример демонстрирует технику использования коллекций. Из него видно, что **добавить элемент в коллекцию можно методом add(...)** класса **ArrayList** и при этом мы нигде не указываем размер коллекции.

В Java коллекции объектов разбиты на три больших категории:

- **List** (список),
- **Set** (множество),
- **Map** (отображение).

List – это список объектов. Объекты можно **добавлять** в список (метод **add()**), **заменять** в списке (метод **set()**), **удалять** из списка (метод **remove()**), **извлекать** (метод **get()**). Существует также возможность организации прохода по списку при помощи итератора.

Set – множество объектов. Те же возможности, что и у List, но объект может входить в множество только один раз. Т.е. двойное добавление одного и того же объекта в множество не изменяет само множество.

Map – отображение или ассоциативный массив. В Map мы добавляем не отдельные объекты, а пары объектов (ключ, значение). Соответственно есть операции поиска значения по ключу. Добавление пары с уже существующим в Map ключом приводит к замене, а не к добавлению. Из отображения (Map) можно получить множество (Set) ключей и список (List) значений.

Коллекции – это наборы произвольных объектов.

Общим свойством коллекций является то, что они работают с Object (базовый класс для всех классов Java). Это означает, что мы можем добавлять в коллекцию любые объекты Java, т.к. Object является базовым классом для всех классов Java. При извлечении из коллекции мы тоже получаем Object. Зачастую это приводит к необходимости преобразования полученного из коллекции объекта к нужному классу.

Это очень удобно для реализации самих коллекций, но не всегда хорошо при программировании. Если мы составляем коллекцию из объектов класса A и в результате ошибки поместим туда объект класса B, то при выполнении программы мы получим ClassCastException при попытке преобразования извлеченного из коллекции элемента к классу A.

Идеальный вариант это, во-первых, получить сообщение об ошибке в точке ее возникновения, а не в точке ее проявления, а, во-вторых, во время трансляции программы, а не при ее выполнении.

8.2 Итераторы

В коллекциях широко используются итераторы. В Java **итератор** – это вспомогательный объект, используемый для прохода по коллекции объектов. Как и сами коллекции, **итераторы базируются на интерфейсе**. Это **интерфейс Iterator**, определенный в пакете **java.util**. Т.е. любой итератор, как бы он не был устроен, имеет следующие три метода:

- **boolean hasNext()** – проверяет есть ли еще элементы в коллекции;
- **Object next()** – выдает очередной элемент коллекции;
- **void remove()** – удаляет последний выбранный элемент из коллекции.

Кроме **Iterator** есть еще **ListIterator** – это расширенный вариант итератора с доп. возможностями и **Enumerator** – это устаревший вариант, оставленный для совместимости с предыдущими версиями.

В свою очередь **интерфейс Collection** имеет **метод *Iterator iterator();***

Это обязывает все классы коллекций создавать поддержку итераторов (обычно реализованы с использованием inner-классов, удовлетворяющих интерфейсу **Iterator**).

Отредактируем листинг 8.1 с использованием интерфейса Iterator (Листинг 8.2)

Листинг 8.2

```
public String toString() {
    String res = "";
    Iterator iter = lst.iterator();
    for(int i = 0; iter.hasNext(); i++) {
       if( i%6 == 0 )
          res += "\n";
       res += " " + iter.next().toString();   // !!!
    }
    return res;
}
```

Метод remove()

В интерфейсе **Iterator** определен метод **remove().** Он позволяет удалить из коллекции последний извлеченный из нее объект. В то же время, в интерфейсе **Collection** (а это значит – во всех классах-коллекциях) есть методы **remove(Object o)** и **removeAll()**.

Здесь нет двойственности возможностей. Дело в том, что удаление из коллекции объектов и вообще модификация коллекции параллельно с проходом по коллекции через итератор запрещено (выдается ConcurrentModificationException). Единственное, что разрешено – это удалить объект методом **remove() из Iterator** (причем, только один - последний извлеченный).

8.3 Классы реализации коллекций

Рассмотрим конкретные реализации коллекций, т.е. классы, которые обеспечивают описанную выше функциональность.

Коллекции-списки (List)

Реализованы в следующий трех вариантах **ArrayList, LinkedList, Vector**.

Класс Vector – это устаревший вариант, оставлен для совместимости с предыдущими версиями.

Класс ArrayList – используется чаще всего. Имеет **два конструктора**:

– **public ArrayList()** // Конструктор пустого списка
– **public ArrayList(Collection c)** // Строит список из любой коллекции

А также следующие методы:

– **public int size()** //Возвращает количество элементов списка;
– **public boolean isEmpty()** //Проверяет, что список пуст;
– **public boolean contains(Object elem)** //Проверяет, принадлежит ли заданный объект списку;
– **public int indexOf(Object elem)** //Ищет первое вхождение заданного объекта в список и возвращает его индекс. Возвращает -1, если объект не принадлежит списку.
– **public int lastIndexOf(Object elem)** //То же самое, но ищет последнее вхождение;
– **public Object clone()** //Создает "поверхностную" копию списка;
– **public Object[] toArray()** //Преобразует список в массив;
– **public Object get(int index)** //Возвращает элемент коллекции с заданным номером;
– **public Object set(int index, Object element)** //Заменяет элемент с заданным номером;
– **public boolean add(Object o)** //Добавляет заданный объект в конец списка;
– **public void add(int index, Object element)** //Вставляет элемент в указанную позицию списка;
– **public Object remove(int index)** //Удаляет заданный элемент списка;
– **public void clear()** //Полностью очищает список;
– **public boolean addAll(Collection c)** //Добавляет к списку (в конец) все элементы заданной коллекции;
– **public boolean addAll(int index, Collection c)** //Вставляет в список с указанной позиции все элементы коллекции;
– **protected void removeRange(int fromIndex, int toIndex)** //Удаляет из коллекции объекты в заданном интервале индексов (исключая toIndex);

Класс LinkedList

LinkedList имеет практически ту же функциональность, что и **ArrayList,** и отличается от него только **способом реализации** и, как следствие, эффективностью выполнения тех или иных операций. Так добавление в **LinkedList** осуществляется в среднем быстрее, чем в **ArrayList**, последовательный проход по списку практически столь же эффективен, как у **ArrayList**, а произвольное извлечение из списка медленнее, чем у **ArrayList**.

1 Сортировка и упорядочивание. Интерфейсы Comparable и Comparator

Начиная с версии 1.5, в Java появились два интерфейса **java.lang.Comparable** и **java.util.Comparator**. Объекты классов, реализующие один из этих интерфейсов, могут быть упорядоченными.

Интерфейс Comparable

В **интерфейсе Comparable** объявлен всего один метод **compareTo(Object obj)**, предназначенный для реализации **упорядочивания объектов класса**. Его удобно использовать при сортировке упорядоченных списков или массивов объектов.

Данный метод **сравнивает** вызываемый объект с **obj**. В отличие от метода **equals**, который возвращает true или false, **compareTo** возвращает:

– 0, если значения равны;
– Отрицательное значение, если вызываемый объект меньше параметра;
– Положительное значение, если вызываемый объект больше параметра.

Метод может выбросить исключение ClassCastException, если типы объектов не совместимы при сравнении.

Аргумент метода **compareTo** имеет тип сравниваемого объекта класса.

Классы Byte, Short, Integer, Long, Double, Float, Character, String уже реализуют **интерфейс Comparable**.

В листинге 8.3 приведен пример реализующий интерфейс.

Листинг 8.3

```
import java.util.TreeSet;
class Comp implements Comparable {
    String str;
    int number;
    Comp(String str, int number) {
        this.str = str;
        this.number = number;
    }
    @Override
    public int compareTo(Object obj) {
        Comp entry = (Comp) obj;
        int result = str.compareTo(entry.str);
        if(result != 0) {
            return result;
        }
        result = number - entry.number;
        if(result != 0) {
            return (int) result / Math.abs( result );
        }
        return 0;
    }
}
public class Example {
    public static void main(String[] args) {
        TreeSet<Comp> ex = new TreeSet<Comp>();
        ex.add(new Comp("Stive Global", 121));
        ex.add(new Comp("Stive Global", 221));
        ex.add(new Comp("Nancy Summer", 3213));
        ex.add(new Comp("Aaron Eagle", 3123));
        ex.add(new Comp("Barbara Smith", 88786));
        for(Comp e : ex) {
```

```
                System.out.println("Str: " + e.str + ", number: " + e.number);
            }
        }
    }
    /*результат:
    * Str: Aaron Eagle, number: 3123
    * Str: Barbara Smith, number: 88786
    * Str: Nancy Summer, number: 3213
    * Str: Stive Global, number: 121
    * Str: Stive Global, number: 221
    */
```

В данном примере, значения сортируются сначала по полю str, а затем по number. Это хорошо видно по двум последним строкам результата.

Для того чтобы сделать **сортировку в обратном порядке**, необходимо внести некоторые изменения в **метод compa**reTo:

```
    @Override
    public int compareTo(Object obj) {
        Comp entry = (Comp) obj;
        int result = entry.str.compareTo(str); // значения меняются местами
        if(result != 0) {
            return result;
        }
        result = entry.number - number; // значения меняются местами
        if(result != 0) {
            return (int) result / Math.abs( result );
        }
        return 0;
    }
    /* результат:
    * Str: Stive Global, number: 221
    * Str: Stive Global, number: 121
    * Str: Nancy Summer, number: 3213
    * Str: Barbara Smith, number: 88786
    * Str: Aaron Eagle, number: 3123
     */
```

Как видно данные отсортированы в обратном порядке, сначала по имени, а потом по цифрам.

Интерфейс Comparator

В интерфейсе Comparator объявлено **два метода compare(Object obj1, Object obj2)** и **equals(Object obj)**.

Метод compare(Object obj1, Object obj2) – так же, как и **метод compareTo** интерфейса Comparable, **упорядочивает объекты класса**. Точно так же на выходе получает 0, положительное значение и отрицательное значение.

Метод может выбросить исключение ClassCastException, если типы объектов не совместимы при сравнении.

Основным отличием **интерфейса Comparator** от **Comparable** является то, что вы можете создавать **несколько видов независимых сортировок**.

В листинге 8.4 приведен пример реализующий интерфейс.

Листинг 8.4

```
public class Product {
   private String name;
   private double price;
   private int quantity;

   public void setName(String name) {
        this.name = name;
   }
   public String getName() {
      return name;
   }
   public double getPrice() {
        return price;
   }
   public void setPrice(double price) {
        this.price = price;
   }
   public void setQuantity(int quantity) {
      this.quantity = quantity;
   }
   public int getQuantity() {
                return quantity;
         }
}
// теперь реализуем интерфейс Comparator, для сортировки по названию
class SortedByName implements Comparator<Product> {
      public int compare(Product obj1, Product obj2) {
          String str1 = ((Product) obj1).getName();
          String str2 = ((Product) obj2).getName();
          return str1.compareTo(str2);
      }
}
// а так же реализуем интерфейс Comparator, для сортировки по цене
class SortedByPrice implements Comparator<Product> {
      public int compare(Product obj1, Product obj2) {
          double price1 = ((Product) obj1).getPrice();
          double price2 = ((Product) obj2).getPrice();
          if(price1 > price2) {
                return 1;
          }
          else if(price1 < price2) {
                return -1;
          }
          else {
                return 0;
          }
```

```
    }
}
// ну и собственно работа с нашими данными
public class ExampleProduct {
    public static void main(String[] args) {
        Product[] p = new Product[3];
        // заполним объект Product содержимым
        p[0] = new Product();
        p[0].setName("Milk");
        p[0].setPrice(7.56);
        p[0].setQuantity(56);

        p[1] = new Product();
        p[1].setName("Coffee");
        p[1].setPrice(17.00);
        p[1].setQuantity(32);

        p[2] = new Product();
        p[2].setName("Tea");
        p[2].setPrice(12.50);
        p[2].setQuantity(0);

        // выведем данные без сортировки
        System.out.println("==== no sorted ====");
        for(Product i : p) {
            System.out.println("Name: " + i.getName() +
           " price: " + i.getPrice() +" quantity: " + i.getQuantity());
        }
        // отсортируем и выведем данные по цене
        System.out.println("====sorted by price====");
        Arrays.sort(p, new SortedByPrice());
        for(Product i : p) {
            System.out.println("Name: " + i.getName() +
           " price: " + i.getPrice() +" quantity: " + i.getQuantity());
        }
        // отсортируем и выведем данные по названию
        System.out.println("====sorted by name ====");
        Arrays.sort(p, new SortedByName());
        for(Product i : p) {
            System.out.println("Name: " + i.getName() +
           " price: " + i.getPrice() +" quantity: " + i.getQuantity());
        }
    }
}
/* результат:
============ no sorted ==============
Name: Milk price: 7.56 quantity: 56
Name: Coffee price: 17.0 quantity: 32
```

Name: Tea price: 12.5 quantity: 0
========== sorted by price==========
Name: Milk price: 7.56 quantity: 56
Name: Tea price: 12.5 quantity: 0
Name: Coffee price: 17.0 quantity: 32
========== sorted by name ==========
Name: Coffee price: 17.0 quantity: 32
Name: Milk price: 7.56 quantity: 56
Name: Tea price: 12.5 quantity: 0
***/**

8.2. Содержание отчета

Отчет о выполнении лабораторной работы должен включать:

1. Теоретические сведения о коллекциях.
2. Диаграмму UML, отражающую созданные классы и типы отношений между ними.
3. Распечатанный листинг выполнения программы, отражающий все этапы ее выполнения.
4. Выводы о выполненной работе.

8.3. Варианты заданий для самостоятельной работы

Вариант 1). Записная книжка контактов. Реализовать сортировку по дате и по времени.

Вариант 2). Система управления доставкой товара. Реализовать сортировку по дате и по времени.

Вариант 3). Телепрограмма. Реализовать сортировку по наименованию передачи и по времени.

Вариант 4). Гостиница. Реализовать сортировку по цене и по количеству мест (человек).

Вариант 5). Реализация готовой продукции. Реализовать сортировку по наименованию и по цене.

Вариант 6). Успеваемость студентов ВУЗА. Реализовать сортировку по фамилиям студентов и по балу ЭГЕ.

Вариант 7). Факультеты. Реализовать сортировку по наименованию факультетов и по номерам корпусов.

Вариант 8). Супермаркеты. Реализовать сортировку по цене и по стране.

Вариант 9). Военный состав. Реализовать сортировку по фамилиям и по зарплатам.

Вариант 10). Учет литературы. Реализовать сортировку по наименованию и по году издательства.

Вариант 11). Учет продажи путевок. Реализовать сортировку по дате и по названиям пансионатов.

Вариант 12). Учет выполненных работ станции техобслуживания. Реализовать сортировку по виду ремонта и по сумме ремонта.

Вариант 13). Медицинское обслуживание пациентов. Реализовать сортировку по наименованию поликлиник и по дате осмотра.

Вариант 14). Библиотека. Реализовать сортировку по фамилиям читателей и по количеству источников литературы в залах.

Вариант 15). Продажа автомобилей. Реализовать сортировку по марке автомобилей и по цене.

Вариант 16). Интернет-магазин. Реализовать сортировку по названию товара и по дате продажи.

Вариант 17). Больница. Реализовать сортировку по названиям отделений и по стоимости лечения.

Вариант 18). Военно-морские учения. Реализовать сортировку по наименованию и по количеству личного состава.

Вариант 19). Туристические туры. Реализовать сортировку по названиям туров и по цене.

Вариант 20). Представления цирка. Реализовать сортировку по городам и датам премьер.

Вариант 21). Аптека. Реализовать сортировку по наименованиям лекарств и по цене.

Вариант 22). Абонентская плата. Реализовать сортировку по фамилиям абонентов и по сумме оплаты.

Вариант 23). Стоматология. Реализовать сортировку по фамилиям пациентов и по сумме оплаты.

Вариант 24). Техобслуживание подвижного состава. Реализовать сортировку по названию и по сумме ремонта.

Вариант 25). Кафедра. Реализовать сортировку по названию кафедр и по стажу работу преподавателей.