Частное учреждение образования
«Колледж бизнеса и права»

УТВЕРЖДАЮ
Ведущий
методист колледжа
_____Е.В. Паскал
«___»_____2021

Специальность: 2-40 01 01 «Программное обеспечение информационных технологий»	Учебная дисциплина: «Основы кроссплатформенного программирования»

Лабораторная работа № 5
Инструкционно-технологическая карта

Тема: «Обобщённые типы и коллекции значений в языке Java»

Цель: Научиться работать с обобщенными типами и коллекциями значений, в частности с интерфейсом Collection и абстрактными классами.

Время выполнения: 4 часа

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретические сведения;
2. Ответить на контрольные вопросы;
3. Откомпилировать примеры программ из раздела «Теоретические сведения»;
4. Выполнить ИДЗ.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ ДЛЯ ВЫПОЛНЕНИЯ РАБОТЫ

Обобщения или generics (обобщенные типы и методы) позволяют нам уйти от жесткого определения используемых типов. Рассмотрим проблему, в которой они нам могут понадобиться.

Допустим, мы определяем класс для представления банковского счета. К примеру, он мог бы выглядеть следующим образом:

```
1    class Account{
2
3        private int id;
4        private int sum;
5
6        Account(int id, int sum){
7            this.id = id;
8            this.sum = sum;
```

```
    }

    public int getId() { return id; }
    public int getSum() { return sum; }
    public void setSum(int sum) { this.sum = sum; }
}
```

Класс Account имеет два поля: id - уникальный идентификатор счета и sum - сумма на счете.

В данном случае идентификатор задан как целочисленное значение, например, 1, 2, 3, 4 и так далее. Однако также нередко для идентификатора используются и строковые значения. И числовые, и строковые значения имеют свои плюсы и минусы. И на момент написания класса мы можем точно не знать, что лучше выбрать для хранения идентификатора - строки или числа. Либо, возможно, этот класс будет использоваться другими разработчиками, которые могут иметь свое мнение по данной проблеме. Например, в качестве типа id они захотят использовать какой-то свой класс.

И на первый взгляд мы можем решить данную проблему следующим образом: задать id как поле типа Object, который является универсальным и базовым суперклассом для всех остальных типов:

```
public class Program{
    public static void main(String[] args) {
        Account acc1 = new Account(2334, 5000); // id - число
            int acc1Id = (int)acc1.getId();
            System.out.println(acc1Id);

            Account acc2 = new Account("sid5523", 5000);    // id - строка
            System.out.println(acc2.getId());
        } }
    class Account{
        private Object id;
        private int sum;
        Account(Object id, int sum){
            this.id = id;
            this.sum = sum;
        }
        public Object getId() { return id; }
        public int getSum() { return sum; }
        public void setSum(int sum) { this.sum = sum; }
    }
```

В данном случае все замечательно работает. Однако тогда мы сталкиваемся с проблемой безопасности типов. Например, в следующем случае мы получим ошибку:

```
Account acc1 = new Account("2345", 5000);
int acc1Id = (int)acc1.getId(); // java.lang.ClassCastException
System.out.println(acc1Id);
```

Проблема может показаться искуственной, так как в данном случае мы видим, что в конструктор передается строка, поэтому мы вряд ли будем пытаться преобразовывать ее к типу int. Однако в процессе разработки мы можем не знать, какой именно тип представляет значение в id, и при попытке получить число в данном случае мы столкнемся с исключением java.lang.ClassCastException.

Писать для каждого отдельного типа свою версию класса Account тоже не является хорошим решением, так как в этом случае мы вынуждены повторяться.

Эти проблемы были призваны устранить обобщения или generics. Обобщения позволяют не указывать конкретный тип, который будет использоваться. Поэтому определим класс Account как обобщенный:

```
class Account<T>{
    private T id;
    private int sum;

    Account(T id, int sum){
        this.id = id;
        this.sum = sum;
    }

    public T getId() { return id; }
    public int getSum() { return sum; }
    public void setSum(int sum) { this.sum = sum; }
}
```

С помощью буквы T в определении класса class Account<T> мы указываем, что данный тип T будет использоваться этим классом. Параметр T в угловых скобках называется универсальным параметром, так как вместо него можно подставить любой тип. При этом пока мы не знаем, какой именно это будет тип: String, int или какой-то другой. Причем буква T выбрана условно, это может и любая другая буква или набор символов.

После объявления класса мы можем применить универсальный параметр T: так далее в классе объявляется переменная этого типа, которой затем присваивается значение в конструкторе.

Метод getId() возвращает значение переменной id, но так как данная переменная представляет тип T, то данный метод также возвращает объект типа T: public T getId().

Используем данный класс:

```
public class Program{
    public static void main(String[] args) {
        Account<String> acc1 = new Account<String>("2345", 5000);
        String acc1Id = acc1.getId();
        System.out.println(acc1Id);
```

```
6
7           Account<Integer> acc2 = new Account<Integer>(2345, 5000);
8           Integer acc2Id = acc2.getId();
9           System.out.println(acc2Id);
10      }
11  }
12  class Account<T>{
13
14      private T id;
15      private int sum;
16
17      Account(T id, int sum){
18          this.id = id;
19          this.sum = sum;
20      }
21
22      public T getId() { return id; }
23      public int getSum() { return sum; }
24      public void setSum(int sum) { this.sum = sum; }
25  }
```

При определении переменной данного класса и создании объекта после имени класса в угловых скобках нужно указать, какой именно тип будет использоваться вместо универсального параметра. При этом надо учитывать, что они работают только с объектами, но не работают с примитивными типами. То есть мы можем написать Account<Integer>, но не можем использовать тип int или double, например, Account<int>. Вместо примитивных типов надо использовать классы-обертки: Integer вместо int, Double вместо double и т.д.

Например, первый объект будет использовать тип String, то есть вместо T будет подставляться String:

```
1  Account<String> acc1 = new Account<String>("2345", 5000);
```

В этом случае в качестве первого параметра в конструктор передается строка.

А второй объект использует тип int (Integer):

```
1  Account<Integer> acc2 = new Account<Integer>(2345, 5000);
```

Обобщенные интерфейсы

Интерфейсы, как и классы, также могут быть обобщенными. Создадим обобщенный интерфейс Accountable и используем его в программе:

```
1   public class Program{
2
3       public static void main(String[] args) {
4
5           Accountable<String> acc1 = new Account("1235rwr", 5000);
6           Account acc2 = new Account("2373", 4300);
7           System.out.println(acc1.getId());
8           System.out.println(acc2.getId());
9       }
```

```
10  }
11  interface Accountable<T>{
12      T getId();
13      int getSum();
14      void setSum(int sum);
15  }
16  class Account implements Accountable<String>{
17
18      private String id;
19      private int sum;
20
21      Account(String id, int sum){
22          this.id = id;
23          this.sum = sum;
24      }
25
26      public String getId() { return id; }
27      public int getSum() { return sum; }
28      public void setSum(int sum) { this.sum = sum; }
29  }
```

При реализации подобного интерфейса есть две стратегии. В данном случае реализована первая стратегия, когда при реализации для универсального параметра интерфейса задается конкретный тип, как например, в данном случае это тип String. Тогда класс, реализующий интерфейс, жестко привязан к этому типу.

Вторая стратегия представляет определение обобщенного класса, который также использует тот же универсальный параметр:

```
1   public class Program{
2
3       public static void main(String[] args) {
4
5           Account<String> acc1 = new Account<String>("1235rwr", 5000);
6           Account<String> acc2 = new Account<String>("2373", 4300);
7           System.out.println(acc1.getId());
8           System.out.println(acc2.getId());
9       }
10  }
11  interface Accountable<T>{
12      T getId();
13      int getSum();
14      void setSum(int sum);
15  }
16  class Account<T> implements Accountable<T>{
17
18      private T id;
19      private int sum;
20
21      Account(T id, int sum){
```

```
22          this.id = id;
23          this.sum = sum;
24      }
25
26      public T getId() { return id; }
27      public int getSum() { return sum; }
28      public void setSum(int sum) { this.sum = sum; }
29  }
```

Обобщенные методы

Кроме обобщенных типов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например:

```
1   public class Program{
2
3       public static void main(String[] args) {
4
5           Printer printer = new Printer();
6           String[] people = {"Tom", "Alice", "Sam", "Kate", "Bob", "Helen"};
7           Integer[] numbers = {23, 4, 5, 2, 13, 456, 4};
8           printer.<String>print(people);
9           printer.<Integer>print(numbers);
10      }
11  }
12
13  class Printer{
14
15      public <T> void print(T[] items){
16          for(T item: items){
17              System.out.println(item);
18          }
19      }
20  }
```

Особенностью обобщенного метода является использование универсального параметра в объявлении метода после всех модификаторов и перед типом возвращаемого значения.

```
1   public <T> void print(T[] items)
```

Затем внутри метода все значения типа T будут представлять данный универсальный параметр.

При вызове подобного метода перед его именем в угловых скобках указывается, какой тип будет передаваться на место универсального параметра:

```
printer.<String>print(people);
printer.<Integer>print(numbers);
```

Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, прежде всего потому, что они имеют фиксированную длину. Эту проблему в Java решают коллекции. Однако суть не только в гибких по размеру наборах объектов, но в и том, что классы коллекций реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.

Классы коллекций располагаются в пакете java.util, поэтому перед применением коллекций следует подключить данный пакет.

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие:

- Collection: базовый интерфейс для всех коллекций и других интерфейсов коллекций
- Queue: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди
- Deque: наследует интерфейс Queue и представляет функционал для двунаправленных очередей
- List: наследует интерфейс Collection и представляет функциональность простых списков
- Set: также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов
- SortedSet: расширяет интерфейс Set для создания сортированных коллекций
- NavigableSet: расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию
- Map: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection

Эти интерфейсы частично реализуются абстрактными классами:

- AbstractCollection: базовый абстрактный класс для других коллекций, который применяет интерфейс Collection
- AbstractList: расширяет класс AbstractCollection и применяет интерфейс List, предназначен для создания коллекций в виде списков
- AbstractSet: расширяет класс AbstractCollection и применяет интерфейс Set для создания коллекций в виде множеств

- AbstractQueue: расширяет класс AbstractCollection и применяет интерфейс Queue, предназначен для создания коллекций в виде очередей и стеков
- AbstractSequentialList: также расширяет класс AbstractList и реализует интерфейс List. Используется для создания связанных списков
- AbstractMap: применяет интерфейс Map, предназначен для создания наборов по типу словаря с объектами в виде пары "ключ-значение"

С помощью применения вышеописанных интерфейсов и абстрактных классов в Java реализуется широкая палитра классов коллекций - списки, множества, очереди, отображения и другие, среди которых можно выделить следующие:

- ArrayList: простой список объектов
- LinkedList: представляет связанный список
- ArrayQeque: класс двунаправленной очереди, в которой мы можем произвести вставку и удаление как в начале коллекции, так и в ее конце
- HashSet: набор объектов или хеш-множество, где каждый элемент имеет ключ - уникальный хеш-код
- TreeSet: набор отсортированных объектов в виде дерева
- LinkedHashSet: связанное хеш-множество
- PriorityQueue: очередь приоритетов
- HashMap: структура данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение
- TreeMap: структура данных в виде дерева, где каждый элемент имеет уникальный ключ и некоторое значение

Интерфейс Collection

Интерфейс Collection является базовым для всех коллекций, определяя основной функционал:

```
1  public interface Collection<E> extends Iterable<E>{
2      // определения методов
3  }
```

Интерфейс Collection является обобщенным и расширяет интерфейс Iterable, поэтому все объекты коллекций можно перебирать в цикле по типу for-each.

Среди методов интерфейса Collection можно выделить следующие:

- boolean add (E item): добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном - false
- boolean addAll (Collection<? extends E> col): добавляет в коллекцию все элементы из коллекции col. При удачном добавлении возвращает true, при неудачном - false
- void clear (): удаляет все элементы из коллекции
- boolean contains (Object item): возвращает true, если объект item содержится в коллекции, иначе возвращает false
- boolean isEmpty (): возвращает true, если коллекция пуста, иначе возвращает false

- Iterator<E> iterator (): возвращает объект Iterator для обхода элементов коллекции
- boolean remove (Object item): возвращает true, если объект item удачно удален из коллекции, иначе возвращается false
- boolean removeAll (Collection<?> col): удаляет все объекты коллекции col из текущей коллекции. Если текущая коллекция изменилась, возвращает true, иначе возвращается false
- boolean retainAll (Collection<?> col): удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции col. Если текущая коллекция после удаления изменилась, возвращает true, иначе возвращается false
- int size (): возвращает число элементов в коллекции
- Object[] toArray (): возвращает массив, содержащий все элементы коллекции

Все эти и остальные методы, которые имеются в интерфейсе Collection, реализуются всеми коллекциями, поэтому в целом общие принципы работы с коллекциями будут одни и те же. Единообразный интерфейс упрощает понимание и работу с различными типами коллекций. Так, добавление элемента будет производиться с помощью метода add, который принимает добавляемый элемент в качестве параметра. Для удаления вызывается метод remove(). Метод clear будет очищать коллекцию, а метод size возвращать количество элементов в коллекции.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для чего используются обобщенные типы?
2. Опишите первую стратегию реализации интерфейса?
3. Опишите вторую стратегию реализации интерфейса?
4. В каком пакете располагаются все классы коллекций?
5. Что лежит в основе всех коллекций?
6. Дайте определение понятию “коллекция”.
7. Назовите преимущества использования коллекций.
8. Что вы знаете о коллекциях типа List?
9. Что вы знаете о коллекциях типа Set?
10. Что вы знаете о коллекциях типа Queue?
11. Опишите интерфейс Collection.
12. Какой интерфейс расширяется с помощью Collection, какой результат это дает?

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Задание 1. Создайте HashMap, содержащий пары значений - имя игрушки и объект игрушки (класс Product).

Перебрать и распечатать пары значений - entrySet(). Перебрать и распечатать набор из имен продуктов - keySet(). Перебрать и распечатать значения продуктов - values(). Для каждого перебора создать свой метод.

Задание 2. Создать класс Student, содержащий следующие характеристики – имя, группа, курс, оценки по предметам. Создать коллекцию, содержащую объекты класса Student. Написать метод, который удаляет студентов со средним баллом <3. Если средний балл>=3, студент переводится на следующий курс. Напишите метод printStudents(List<Student> students, int course), который получает список студентов и номер курса. А также печатает на консоль имена тех студентов из списка, которые обучаются на данном курсе.

Задание 3. Создайте класс Pet и его наследников Cat, Dog, Parrot. Создайте отображение из домашних животных, где в качестве ключа выступает имя животного, а в качестве значения класс Pet. Добавьте в отображение разных животных. Создайте метод выводящий на консоль все ключи отображения.

Задание 4. Дополните проект задания 2 из прошлой лабораторной работы:

a) Создать список сотрудников с фиксированной и почасовой оплатой. Упорядочить всю последовательность рабочих по убыванию среднемесячной зарплаты. При совпадении зарплаты – упорядочить данные в алфавитном порядке по имени. Вывести идентификатор работника, имя и среднемесячную зарплату для всех элементов списка.

b) Вывести первые 5 имен работников из полученного выше списка.

c) Вывести последние 3 идентификаторы работников из полученного выше списка.

2. ДОМАШНЕЕ ЗАДАНИЕ

Индивидуальное задание

3. ЛИТЕРАТУРА

И. Н. Блинов В. С. Романчик, Java, Четыре четверти, 2020

Преподаватель А.С.Кибисова

Рассмотрено на заседании цикловой комиссии программного обеспечения информационных технологий
Протокол № ____ от «___» _____ 2021
Председатель ЦК _____В.Ю.Михалевич