Методические указания 4

Обобщения.

Коллекции Часть 1

Понятие обобщения. Обобщённые классы, методы и интерфейсы. Наследование обобщённых классов. Ограничения при работе с обобщениями. Введение в коллекции. Интерфейс List и его реализации.

Небольшая задача

Перед нами стоит задача: создать класс, который позволит хранить в себе один объект любого типа. В таком случае мы можем создать класс SimpleBox, у которого будет единственное поле типа Object, в которое можно будет записать объект абсолютно любого типа. Для проверки работы класса напишем немного кода в классе BoxDemoApp:

```
public  class SimpleBox {
private Object obj;
      public Object
getObj() {          return
obj;
    }
     public   void setObj(Object  obj)  {
this .obj = obj;
    }       public
SimpleBox(Object obj) {
this .obj = obj;
    }
}
public  class BoxDemoApp {
     public  static  void main(String[] args) {
SimpleBox intBox1 =  new SimpleBox( 20) ;
SimpleBox intBox2 =  new SimpleBox( 30) ;

         if  (intBox1.getObj()     instanceof  Integer  &&  intBox2.getObj()
instanceof Integer) {
             int sum = (Integer)intBox1.getObj() + (Integer)intBox2.getObj();
System.out.println( "sum = " + sum);
        }  else {
            System.out.println( "Содержимое коробок отличается по типу"
);          }
         // вызываем какой-нибудь метод, которому отдаём intBox1,
         // и этот метод кладёт в коробку String
intBox1.setObj( "Java" );

         // продолжаем наш код и при выполнении получим ClassCastException
int secondSum = (Integer)intBox1.getObj() + (Integer)intBox2.getObj();     }
}
```

В первых строках кода метода main() создаём две коробки intBox1 и intBox2, в которые при инициализации складываем значения 20 и 30. Предположим, нужно извлечь числа из коробок и получить их сумму. Чтобы извлечь из коробки число, можно применить метод getObj(), но он вернёт не int а Object. Следовательно, придётся добавить приведение к типу Integer. Кроме того, чтобы не получить ClassCastException, перед строками с приведением типов желательно делать проверку (instanceof) на то, что в коробках лежат данные нужного типа. В результате мы получим сумму чисел из коробок и напечатаем её в консоль.

Допустим, в коде мы вызываем некий метод, и отдаём туда ссылку на intBox1. Выполняемый метод может положить в нашу коробку всё что угодно, например, строку «Java». И если после этого мы

попытаемся сложить содержимое наших коробок, забыв при этом сделать instanceof, то получим ClassCastException.

Таким образом, можно выделить три проблемы при использовании такого подхода:

- каждый раз, когда мы хотим вытащить данные из нашей универсальной коробки, нам необходимо выполнять приведение типов;
- чтобы не получить ClassCastException, перед каждым приведением типов необходимо делать проверку типов данных с помощью instanceof;
- если мы где-то применим приведение типов и забудем прописать instanceof, то появится вероятность появления ClassCastException в этой части кода.

Для того, чтобы решить эти проблемы, в Java применяются обобщения.

Понятие обобщения

Обобщения — это параметризованные типы, которые позволяют объявлять обобщённые классы, интерфейсы и методы, где тип данных, которыми они оперируют, указан в виде параметра. Обобщения в Java повышают безопасность типов и делают управление проще. Исключается необходимость применять явные приведения типов, так как благодаря обобщениям все приведения выполняются неявно, в автоматическом режиме.

Чтобы понять, почему нам больше не надо делать явное приведение типов и как это влияет на безопасность, рассмотрим простой пример обобщённого класса:

```
public  class TestGeneric<T> {
private T obj;

     public TestGeneric(T obj) {
this .obj = obj;
    }

     public     T     getObj()     {
return obj;
    }

     public  void showType() {
        System.out.println( "Тип  T:  "  +  obj.getClass().getName());
}
}
public  class GenericsDemoApp {
     public  static  void main(String args[]) {
        TestGeneric<String> genStr =  new TestGeneric<>( "Hello" );
genStr.showType();
```

```
        System.out.println( "genStr.getObject(): " +
genStr.getObj());
        TestGeneric<Integer> genInt =  new TestGeneric<>( 140) ;
genInt.showType();
        System.out.println(       "genInt.getObject():       "       +
genInt.getObj());         int valueFromGenInt = genInt.getObject();
        String  valueFromGenString  =  getStr.getObject();
//  genInt.setObj( "Java" ); // Ошибка компиляции !!!     }
}
```

Важно!

- При получении значений из genInt и genStr не требуется преобразование типов: genInt.getObj() сразу возвращает Integer, а genStr.getObj() — String, то есть приведение типов выполняется неявно и автоматически.
- Если объект создан как TestGeneric<Integer> genInt, то мы не сможем записать в него строку. При попытке написать, например, genInt.setObj("Java") мы получим ошибку на этапе компиляции, то есть обобщения отслеживают корректность используемых типов данных.

Эти две особенности использования обобщений повышают безопасность типов данных, и упрощают написание кода.

Результат выполнения:

```
Тип T: java.lang.String
genStr.getObject(): Hello
Тип T: java.lang.Integer
genInt.getObject():  140
```

T в объявлении класса public class TestGeneric<T> — это имя параметра типа, на место которого при создании объекта класса TestGeneric будет подставлен конкретный тип данных. У объекта genStr из примера выше T = String, а для genInt T = Integer. То есть, используя обобщения, мы можем создавать переменные типы данных.

Важно! Обобщения работают только со ссылочными типами данных. Для работы с примитивами надо будет использовать классы-обёртки.

T в объявлении обобщённого класса — всего лишь имя переменного типа, вместо неё можно использовать любую другую букву (E, N, V, …).

Ссылка на одну специфическую версию обобщённого типа не обладает совместимостью с другой версией того же обобщённого типа. Например:

```
TestGeneric<String> genStr =  new TestGeneric<>( "Hello"
);  TestGeneric<Integer> genInt =  new TestGeneric<>( 140)
; genInt = genStr;  // Ошибка
```

Объекты genInt и genStr принадлежат классу TestGeneric<T>, но представляют собой ссылки на разные типы — ведь типы их параметров отличаются.

Обобщённый класс с несколькими параметрами типа

Для обобщённого типа можно объявлять более одного параметра, используя список, разделённый запятыми:

```
public  class TwoGen<T, V>
{     private T obj1;
private V obj2;
     public TwoGen(T obj1, V
obj2) {          this .obj1 =
obj1;          this .obj2 = obj2;
   }
    public  void showTypes() {
       System.out.println(        "Тип        T:        "        +
obj1.getClass().getName());        System.out.println( "Тип V: "
+ obj2.getClass().getName());      }        public  T  getObj1()  {
return obj1;
   }
    public   V   getObj2()   {
return obj2;
   }
}
public  class SimpleGenApp {
   public  static  void main(String args[]) {
    TwoGen<Integer, String> twoGenObj =  new TwoGen<Integer, String>(
555,
"Hello" );
    twoGenObj.showTypes();      int
intValue = twoGenObj.getObj1();
String strValue = twoGenObj.getObj2();
    System.out.println(intValue);
System.out.println(strValue);
  }
}
```

Ограниченные типы

В примерах, рассмотренных выше, параметры типов можно было заменить любыми типами классов. Но иногда бывает нужно ограничить набор этих перечень типов.

Создадим обобщённый класс, который содержит в себе массив (мы предполагаем что это будет массивом чисел любого типа) и метод, возвращающий среднее значение этого массива.

```
public  class Stats<T> {
private T[] nums;

    public Stats(T... nums) {
this .nums = nums;
   }      public  double avg() {
double sum =  0.0;          for (int  i =
0; i < nums.length; i++) {
           sum += nums[i];  // Ошибка
       }         return sum /
nums.length;
   }
}
```

В методе avg() мы создаём временную double-переменную sum и пытаемся с помощью неё сложить все числа, лежащие в массиве nums. Но первая проблема заключается в том, что Java видит, что мы пытаемся складывать объекты (хотя мы собираемся хранить в этом массиве только числа), и выдаёт ошибку. Вторая же проблема в том, что, если даже это будет массив Integer, то у каждого объекта при суммировании надо будет вызывать метод doubleValue(), который будет преобразовывать каждый объект Integer в примитив double. Давайте попробуем решить обе.

При работе с обобщениями будем использовать ограниченные типы. Когда указывается параметр типа, можно создать ограничение сверху, которое укажет суперкласс, от которого должны быть унаследованы все аргументы типов. Для этого используется ключевое слово extends:

```
<T extends суперкласс>
```

Важно! В роли ограничителя сверху может выступать не только класс, но и один или несколько интерфейсов. Для указания нескольких элементов используется оператор &:

<T extends Cat>
<T extends Animal & Serializable>
<T extends Serializable>
<T extends Cloneable & Serializable>

Обратите внимание, что даже если вы ограничиваете T интерфейсом, всё равно используется ключевое слово extends.

Если в качестве ограничителя используются и класс, и интерфейс, то класс должен быть указан первым.

Это означает, что параметр T может быть заменен только самим суперклассом или его подклассами. Он объявляет включающую верхнюю границу. Можно использовать ограничение сверху, чтобы исправить класс Stats, указав класс Number как верхнюю границу используемого параметра типа. Посмотрим, что нам это даст:

```
public  class Stats<T  extends Number> {
```

```
    private T[] nums;

    public Stats(T... nums) {
this .nums = nums;
   }      public
double avg() {
double sum =  0.0;
        for (int  i =  0; i < nums.length; i++) {
            // У nums[i] появился метод doubleValue() из класса Number,
// который позволяет любой числовой объект привести к double          sum
+= nums[i].doubleValue();
       }         return sum /
nums.length;
   }
}
public  class StatsDemoApp {
   public  static  void main(String args[]) {
    Stats<Integer> intStats =  new Stats<Integer>( 1,   2,   3,   4,   5) ;
    System.out.println( "Ср. знач. intStats равно " + intStats.avg());
    Stats<Double> doubleStats =  new Stats<Double>( 1.0,   2.0,   3.0,   4.0,
5.0) ;
    System.out.println( "Ср. знач. doubleStats равно " + doubleStats.avg());
     // Это не скомпилируется, потому что String не является подклассом
Number
     // Stats<String> strStats = new Stats<>("1", "2", "3", "4", "5");
     // System.out.println("Ср. знач. strStats равно " + strStats.avg());
}
}
```

Объявление public class Stats<T extends Number> сообщает компилятору, что все объекты типа T являются подклассами класса Number и поэтому могут вызывать метод doubleValue(), как и любой другой из класса Number. Ограничивая параметр T, мы предотвращаем создание нечисловых объектов класса Stats.

Использование метасимвольных аргументов

Давайте попробуем добавить в класс Stats метод sameAvg(), который будет проверять равенство средних значений массивов двух объектов типа Stats независимо от того, какого типа числовые значения в них содержатся. Допустим, в одном будет new int[] { 1, 2, 3 }, а во втором — new double { 1.0, 2.0, 3.0 }.

Метод sameAvg() должен принимать на вход объект типа Stats и сравнивать его среднее значение со средним значением вызывающего объекта. Код применения этого метода может выглядеть вот так:

```
public  class StatsDemoApp {
    public  static  void main(String args[]) {
      Stats<Integer> intStats =  new Stats<>( 1,   2,   3,   4,   5) ;
```

```
      Stats<Double> doubleStats =  new Stats<>( 1.0,    2.0,    3.0,
4.0,   5.0) ;        if (intStats.sameAvg(doubleStats)) {
         System.out.println( "Средние значения равны" );
}  else {
         System.out.println( "Средние значения не равны" );
      }
   }
}
```

Класс Stats является обобщённым, и при написании метода sameAvg() возникает вопрос: какой тип указать для аргумента Stats? Попробуем использовать обобщённый тип T:

```
public  class Stats<T  extends Number> {
// ...
    public  boolean sameAvg(Stats<T> another) {
return Math.abs( this .avg() - another.avg()) <  0.0001;
}
    // ...
}
```

Заметка. Чтобы не столкнуться с ошибкой округления при сравнении двух дробных чисел, мы сравниваем средние значения в пределах дельты 0.0001.

Такой код будет работать только с объектом класса Stats, тип которого совпадает с вызывающим объектом. Если вызывающий объект имеет тип Stats<Integer>, то параметр another обязательно должен принадлежать к аналогичному типу.

```
public  class StatsDemoApp {
    public  static  void main(String args[]) {
      Stats<Integer> intStats1 =  new Stats<>( 1,   2,   3,   4,   5) ;
      Stats<Integer> intStats2 =  new Stats<>( 2,   1,   3,   4,   5) ;
      Stats<Double> doubleStats =  new Stats<>( 1.0,   2.0,   3.0,   4.0,
5.0) ;
      System.out.println(intStats1.sameAvg(intStats2));       // Так
работает
       // System.out.println(intStats1.sameAvg(doubleStats)); // Ошибка
// (T = Integer) != (T = Double)
   }
}
```

Чтобы создать обобщённую версию метода sameAvg(), следует использовать метасимвольные аргументы. Это средство обобщений Java, которое обозначается символом «?» и представляет собой неизвестный тип.

```
public  class Stats<T  extends Number> {
// ...     public  boolean sameAvg(Stats<?>
another) {
       return Math.abs( this .avg() - another.avg()) <
0.0001;     }
    // ...
}
```

Stats<?> соответствует любому объекту класса Stats. Можно сравнивать средние значения любых двух объектов этого класса.

```
public  class Stats<T  extends Number> {
private T[] nums;
     public Stats(T[]
nums) {          this .nums
= nums;
   }      public
double avg() {
double sum =  0.0;
        for ( int i =  0; i < nums.length; i++)
{                  sum += nums[i].doubleValue();
       }
        return sum / nums.length;
   }
    public  boolean sameAvg(Stats<?> another) {
        return Math.abs( this .avg() - another.avg()) <
0.0001;     }
}
public  class WildcardDemoApp {
   public  static  void main(String args[]) {
    Stats<Integer> iStats =  new Stats<>(1, 2, 3, 4, 5);
    System.out.println( "Среднее iStats = " +
iStats.avg());
    Stats<Double> dStats =  new Stats<>(1.1, 2.2, 3.3, 4.4,
5.5);          System.out.println( "Среднее dStats = " +
dStats.avg());
    Stats<Float> fStats =  new Stats<>(1.0f, 2.0f, 3.0f, 4.0f,
5.0f);     System.out.println( "Среднее fStats = " + fStats.avg());
    System.out.print( "Средние iStats и dStats " );
     if (iStats.sameAvg(dStats)) {
System.out.println( "равны" );
    }  else {
      System.out.println( "отличаются"
);     }

    System.out.print( "Средние iStats и fStats" );
     if (iStats.sameAvg(fStats)) {
System.out.println( "равны" );
    }  else {
      System.out.println( "отличаются"
);     }


  }
}
```

Результат работы программы:

```
Среднее iStats =  3.0
Среднее dStats =  3.3
Среднее fStats =  3.0
Средние iStats и dStats отличаются
Средние iStats и fStats равны
```

Метасимвольный аргумент не влияет на тип создаваемого объекта класса Stats. Это зависит от extends в объявлении класса Stats.

Ограничения

Ограничения на статические члены

Никакой статический член не может использовать тип параметра, объявленный в его классе:

```
public  class WrongGenericClass<T> {    static T data;
// Неверно, нельзя создать статические
                                        // переменные типа T
   static T getData() {  return data; }  // Неверно, ни один статический
метод                                        // не может использовать T }
```

Нельзя объявить статические члены, использующие обобщённый тип. Но можно объявлять обобщённые статические методы, определяющие их собственные параметры типа.

Ограничения обобщённых исключений

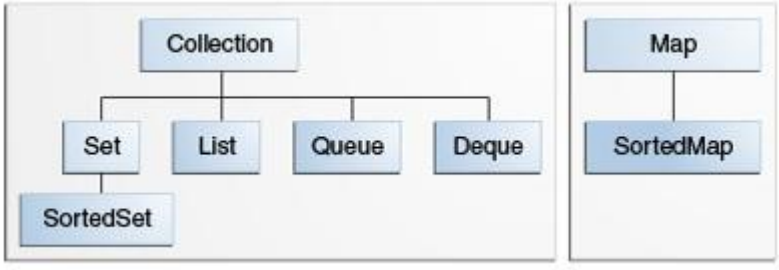
Обобщённый класс не может расширять класс Throwable. Значит, создать обобщённые классы исключений невозможно.

Коллекции

Коллекция представляет собой контейнер, который хранит в себе множество элементов, и позволяющий работать с этой группой элементов как с одним объектом. Коллекции используются для хранения, получения, обработки множества объектов.

Если вы только начинаете программировать, то можете сразу пролистать методичку до пункта “класс ArrayList”. Если уже научились работать с коллекциями, то можете дополнить ваши знания тем что здесь написано.

При работе с коллекциями в Java используется следующий набор интерфейсов: Collection, Set, SortedSet, List, Queue, Deque, Map, SortedMap. Иерархия этих интерфейсов представлена на рисунке ниже (https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html) .



Давайте посмотрим на основное назначение каждого интерфейса:

- **Collection** - является “корневый” элементом иерархии коллекций. Коллекция представляет собой группу объектов, называемых элементами. Интерфейс описывает функционал присущий абсолютно всем коллекциям.
- **Set** - коллекция, представляющая собой множество элементов, которое не может содержать дубликаты.
- **List** - коллекция, содержащая в себе последовательность элементов. В этом типе коллекций дубликаты разрешены. List позволяет обращаться к своим элементам по индексу, добавлять/изменять/удалять элементы.
- **Queue** - коллекция, позволяющая управлять процессом обработки элементов. Как правило, работает в режиме FIFO (first-in, first-out, первый вошел - первый вышел, например, очередь в кабинет). В таком случае элементы добавляются в хвост, а забираются с головы.
- **Deque** - коллекция, позволяющая управлять процессом обработки элементов. Deque может использоваться как в режиме FIFO , так и в режиме LIFO (last-in, first-out, последний вошел - первый вышел, например, в стопку бумаг кладут сверху листы, и оттуда же их забирают). Deque позволяет добавлять/изменять/удалять элементы как в голове, так и в хвосте коллекции.
- **Map** , коллекция которая отображает ключ, к его значению. Map не может содержать дублирующиеся ключи.
- **SortedSet** и **SortedMap** представляют собой упорядоченные в порядке возрастания версии коллекций Set (сортировка идет по элементам) и Map (сортировка производится по ключам).

Перейдем теперь к практическим вопросам работы с коллекциями.

Класс ArrayList

Обычные массивы после создания имеют фиксированную длину, которую нельзя изменить напрямую. Если такая необходимость возникнет, можно попытаться это сделать следующим образом:

```
public  static  void main(String[] args) {
    int [] arr = { 1,   2,   3,
4} ;     int [] arrNew =  new
int [10 ] ;
   System.arraycopy(arr,  0, arrNew,      0,
arr.length);          arr = arrNew;    arrNew =
null ;
}
```

В коде выше создан массив arr имеющий длину 4. Для увеличения его размера до 10 элементов, мы создаем новый массив arrNew длиной 10, с помощью метода System.arraycopy() копируем все элементы из arr в arrNew, и записываем в arr ссылку на новый массив arrNew. В результате arr изменил свой размер с 4 до 10.

Чтобы не “изобретать колесо” для решения этой проблемы, можно воспользоваться готовыми классами Java. Класс ArrayList представляет собой динамический массив, размер которого может изменяться по мере необходимости (в основе своей он использует обычный массив).

Важно! Класс ArrayList, как и все коллекции в целом, может работать только с ссылочными типами данных. Для хранения примитивов необходимо использовать “обертки”: Byte, Short, Integer, Long, Float, Double, Character, Boolean.

У ArrayList есть два основных параметра: размер (size) и емкость (capacity). size указывает на то, сколько объектов уже хранится в нем, а capacity - на сколько объектов рассчитана данная коллекция

(по-умолчанию начальная емкость равна 10, и может задана через конструктор). Схематично структура и принцип работы ArrayList представлены на рисунке 1.

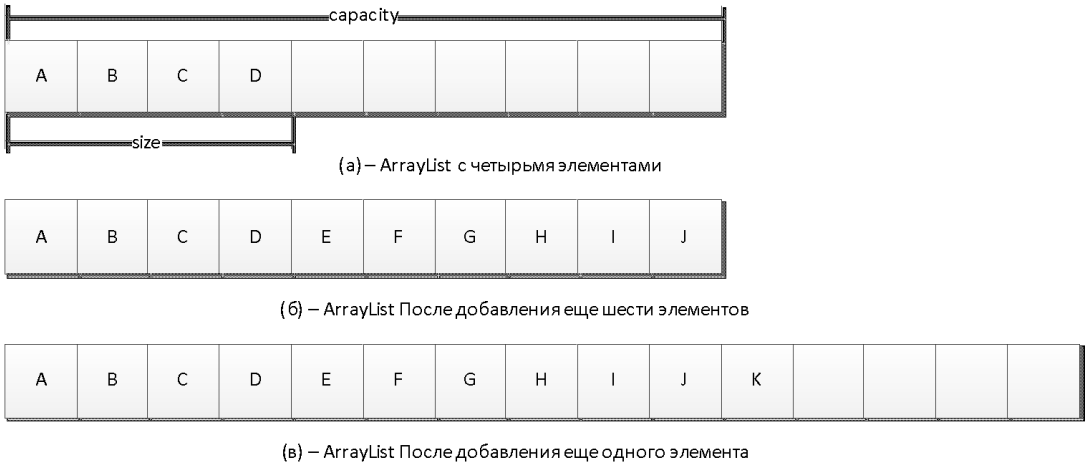


Рисунок 1 — Схема работы ArrayList

При добавлении объектов в коллекцию, ее size растет, как только size попытается превысить capacity, ArrayList расширяется путем увеличения capacity в полтора раза. При расширении, элементы, которые были в коллекции до расширения, сохраняются. При таком подходе пользователю не приходится думать о том, что может закончится место для хранения объектов, особенно в случаях когда при старте программы неизвестно сколько может понадобиться места (допустим считываем строки из файла, содержимое которого может меняться с течением времени).

Еще одной важной особенностью ArrayList является то, что в нем не может быть “пустых мест” между элементами. Допустим существует ArrayList с элементами { 1, 2, 3, 4, 5 }, при удалении элемента 3, он примет вид { 1, 2, 4, 5 }, то есть элементы, после удаленного переместятся влево на одну позицию (то есть на месте 3 не образуется никакого null или чего-либо еще). null может содержаться в ArrayList только если вы его туда добавите. Вновь созданный ArrayList также не состоит из набора null, при попытке обратиться к элементу с индексом, превышающим size - 1, будет сгенерировано исключение IndexOutOfBoundException (например, если в ArrayList есть набор элементов {1, 1, 1}, и мы попытаемся обратиться к элементу с индексом 3).

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> collection)	Создает ArrayList, инициализируемый элементами заданной коллекции collection
ArrayList(int initialCapacity)	Создает ArrayList, имеющий указанную начальную емкость (initialCapacity)

Основные методы 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> arrayList =  new
ArrayList<>();     arrayList.add( "A" );
arrayList.add( "B" );     arrayList.add( "C" );
arrayList.add( "D" );     arrayList.add( "E" );
arrayList.add( 1,   "A0" );
System.out.println(arrayList);
    arrayList.remove( "E" );
arrayList.remove( 2) ;
    System.out.println(arrayList);
}

// Результат:
// [A, A0, B, C, D, E]
// [A, A0, C, D]
```

Заметка.
Начиная с версии Java 1.8, при объявлении коллекций допустимо использовать запись:
`ArrayList<String> arrayList =  new ArrayList<>();`
До этой версии:
`ArrayList<String> arrayList =  new ArrayList<String>();`
То есть в версии 1.8+ необязательно дублировать <тип> в правой части, и достаточно указать <>.
Если компилятор начинает на это ругаться, проверьте используемую версию языка.

В приведенном выше коде, создается ArrayList, который может хранить в себе только объекты типа String, и в него по-порядку в конец списка добавляются A, B, C, D, E, после чего в ячейку с индексом 1 кладется элемент A0, что приводит к смещению всех элементов после 1-го на одну ячейку вправо. Затем печатаем содержимое `arrayList` в консоль (у коллекций очень удобная перегрузка метода toString(), которая позволяет отдавать их в метод System.out.println() для отображения содержимого). После первой печати, из ArrayList удаляется два элемента, вначале по объекту “E”(в коллекции производится поиск этого элемента и его удаление), а затем удаляется элемент из ячейки с индексом 2. Ну и в конце концов печатается итоговая версия al.

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 выполнения дорогостоящей операции перераспределения памяти.

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(). Схематично принцип работы этого метода показан на рисунке 2.

Рисунок 2 - Принцип работы метода trimToSize()

Получение массива из ArrayList. При работе с ArrayList может возникнуть необходимость преобразовать его в обычный массив, что можно сделать вызвав метод toArray(). Вот некоторые причины для этого: ускорение выполнения некоторых операций, передача массива в качестве параметра методам, которые не перегружены для работы с коллекциями, интеграция кода, использующего коллекции, с кодом, который работает только с массивами. Имеется две перегрузки метода toArray():

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

В первом случае метод toArray() возвращает массив объектов типа Object, а во второй — массив элементов того же типа, что и тип коллекции(например, ArrayList<String> преобразуется к массиву String[], что очень удобно). Рассмотрим пример использования данного метода.

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

Код довольно простой, вначале создаем целочисленный ArrayList и складываем в него значения {1, 2, 3} после чего применяя метод toArray() преобразуем его к массиву того же типа что и сам ArrayList. Как было сказано ранее, для работы с примитивным типом int используем его обертку - Integer.

Важно! Что нужно помнить при работе с ArrayList:

- ArrayList представляет собой динамический массив(список) в Java;
- Получение значения по индексу ячейки осуществляется с помощью метода **get()** ;
- Для добавления элемента в ArrayList используем метод **add()** ;
- Метод set() позволяет заменить значение ячейки по ее индексу;
- Чтобы удалить элемент по индексу или значению, используем метод remove();
- При удалении элемента не с конца ArrayList, все элементы, идущие после удаляемого элемента, будут смещены на 1 позицию влево.

Класс LinkedList

LinkedList<E> предоставляет структуру данных связного списка, где E обозначает ссылочный тип хранимых объектов. У класса LinkedList имеется два конструктора: LinkedList() - создает пустой LinkedList; и LinkedList(Collection<? extends E> collection), позволяющий создать LinkedList, который будет содержать элементы другой коллекции.

Структура LinkedList значительно отличается от ArrayList. Каждый элемент в связном списке имеет ссылку на предыдущий и на следующий элементы. Сам же LinkedList имеет ссылку на свой первый и последний элемент. Структура LinkedList схематично показана на рисунке 3.

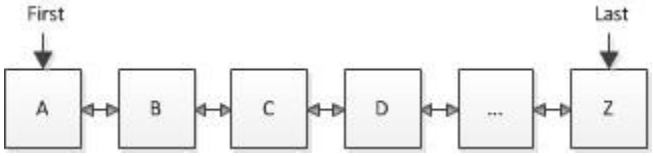


Рисунок 3 — Структура LinkedList

LinkedList, как и ArrayList, имеет параметр size, показывающий количество элементов в списке, но не имеет параметра capacity, так как все объекты связаны между собой ссылками друг на друга и могут быть раскиданы по памяти.

LinkedList позволяет производить поиск элемента по индексу, однако из-за своей структуры, для поиска элемента приходится обходить элементы коллекции или от первого, или от последнего, и пробегать по ссылкам, пока не будет найден запрошенный элемент. Выбор начала поиска зависит от указанного индекса, и size объекта LinkedList, если индекс меньше size / 2, то поиск будет начинаться с начала коллекции, в противном случае - с конца. Это позволяет уменьшить количество операций необходимых для поиска.

При удалении элемента нет необходимости куда-либо смещать элементы коллекции (как это происходит в ArrayList), достаточно лишь переписать ссылки у двух соседних с удаленным объектов. При большом количестве элементов в коллекции, удаление из начала LinkedList будет производиться быстрее чем из начала ArrayList, это связано опять же с тем, что LinkedList перепишет всего пару ссылок, а ArrayList будет перемещать все элементы после удаленного влево на одну позицию.

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> linkedList =  new
LinkedList<>();     linkedList.add( "F" );
linkedList.add( "B" );     linkedList.add( "D" );
linkedList.add( "E" );     linkedList.add( "C" );
linkedList.addLast( "Z" );     linkedList.addFirst(
"A" );     linkedList.add( 1,   "A2" );
    System.out.println( "1. linkedList: " +
linkedList);     linkedList.remove( "F" );
linkedList.remove( 2) ;
    System.out.println( "2. linkedList: " +
linkedList);     linkedList.removeFirst();
linkedList.removeLast();
    System.out.println( "3. linkedList: " + linkedList);
    String val = linkedList.get( 2) ;
linkedList.set( 2, val +      " изменено" );
    System.out.println( "4. linkedList: " +
linkedList); }

// Результат:
// 1. linkedList: [A, A2, F, B, D, E, C, Z]
// 2. linkedList: [A, A2, D, E, C, Z]
// 3. linkedList: [A2, D, E, C]
// 4. linkedList: [A2, D, E изменено, C]
```

Обратите внимание, как третий элемент связного списка linkedList изменяется с помощью методов get() и set(). Для получения значения элемента на позиции n, выполняется метод get(n). Для присвоения нового значения элементу на этой позиции, методу set() передаётся соответствующий индекс и новое значение.

Практическое задание

1. Написать метод, который меняет два элемента массива местами (массив может быть любого ссылочного типа).
2. Написать метод, который преобразует массив в ArrayList.
3. Задача:
    a. Даны классы Fruit -> Apple, Orange.
    b. Класс Box, в который можно складывать фрукты. Коробки условно сортируются по типу фрукта, поэтому в одну коробку нельзя сложить и яблоки, и апельсины.
    c. Для хранения фруктов внутри коробки можно использовать ArrayList.
    d. Написать метод getWeight(), который высчитывает вес коробки. Задать вес одного фрукта и их количество: вес яблока — 1.0f, апельсина — 1.5f (единицы измерения не важны).

e. Внутри класса Box написать метод Compare, который позволяет сравнить текущую коробку с той, которую подадут в Compare в качестве параметра. True, если их массы равны, False — в противном случае. Можно сравнивать коробки с яблоками и апельсинами.

f. Написать метод, который позволяет пересыпать фрукты из текущей коробки в другую. Помним про сортировку фруктов: нельзя высыпать яблоки в коробку с апельсинами. Соответственно, в текущей коробке фруктов не остаётся, а в другую перекидываются объекты, которые были в первой.

g. Не забываем про метод добавления фрукта в коробку.

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.: Вильям с , 2015. — 1 376 с.