# Практическая работа № 20. Работа с дженериками.

**Цель** данной практической работы – научиться работать с обобщенными типами в Java и применять их в программах.

## Теоретические сведения

### Понятие дженериков.

Введение в Дженерики. В JDK представлены дженерики (перевод с англ. generics), которые поддерживают абстрагирование по типам (или параметризованным типам). В объявлении классов дженерики представлены обобщенными типами, в то время как пользователи классов могут быть конкретными типами, например во время создания объекта или вызова метода.

Вы, конечно, знакомы с передачей аргументов при вызове методов в языке C++. Когда вы передаете аргументы внутри круглых скобок () в момент вызова метода, то аргументы подставляются вместо формальных параметров с которыми объявлен и описан метод. Схожим образом в generics вместо передаваемых аргументов мы передаем информацию только о типах аргументов внутри угловых скобок <> (так называемая diamond notation или алмазная запись).

Основное назначение использования дженериков — это абстракция работы над типами при работе с коллекциями («Java Collection Framework»).

Например, класс ArrayList можно представить следующим образом для получения типа дженериков <E> следующим образом:

Листинг 20.1 Класс ArrayList с параметризованным типом E

```
public class ArrayList<E> implements List<E> ....
{
        //конструктор
        public ArrayList() { ..... }

        // методы public
        public boolean add(E e) { ..... }
        public void add(int index, E element) { .....
}
        public boolean addAll(int index, Collection<?
extends E> c)
```

```
    public abstract E get(int index) { ..... }
    public E remove(int index)
    .......
}
```

Чтобы создать экземпляр ArrayList, пользователям необходимо предоставить фактический тип для <E> для данного конкретного экземпляра. Фактический тип будет заменять все ссылки на E внутри класса. Например:

Листинг 20.2 Пример создания экземпляра класса ArrayList с типом Integer

```
ArrayList<Integer> lst1 = new ArrayList<Integer>();
// E подстановка Integer
lst1.add(0, new Integer(88));
lst1.get(0);

ArrayList<String> lst2 = new ArrayList<String>();
// E подстановка String
lst2.add(0, "Hello");
lst2.get(0);
```

В приведенном выше примере показано, что при проектировании или определении классов, они могут быть типизированными по общему типу; в то время как пользователи классов предоставляют конкретную фактическую информацию о типе во время создания экземпляра объекта типа класс. Информация о типе передается внутри угловых скобок <>, так же как аргументы метода передаются внутри круглой скобки ().

В этом плане общие коллекции не являются безопасными типами!

Если вы знакомы с классами- коллекциями, например, такими как ArrayList, то вы знаете, что они предназначены для хранения объектов типа java.lang.Object. Используя основной принцип ООП-полиморфизм, любой подкласс класса Object может быть заменен на Object. Поскольку Object является общим корневым классом всех классов Java, то коллекция, предназначенная для хранения Object, может содержать любые классы Java. Однако есть одна проблема. Предположим, например, что вы хотите определить ArrayList из объектов класса String. То при выполнении операций, например операции add(Object) объект класса String будет

неявным образом преобразовываться в Object компилятором. Тем не менее, во время поиска ответственность программиста заключается в том, чтобы явно отказаться от Object обратно в строку. Если вы непреднамеренно добавили объект не-String в коллекцию, то компилятор не сможет обнаружить ошибку, а понижающее приведение типов (от родителя к дочернему) завершится неудачно во время выполнения (будет сгенерировано ClassCastException throw).

Ниже приведен пример:

Листинг 20.3 Пример использования интерфейсной ссылки List для инициализации объектом класса ArrayList

```
import java.util.*;
public class ArrayListWithoutGenericsTest {
   public static void main(String[] args) {
      List strLst = new ArrayList();
// List и ArrayList содержит тип Objects
      strLst.add("alpha");
// Неявное преобразование String в Object
      strLst.add("beta");
      strLst.add("charlie");
      Iterator iter = strLst.iterator();
      while (iter.hasNext()) {
         String str = (String)iter.next();
// необходимо выполнить понижающее преобразование типов Object обратно в String
         System.out.println(str);
      }
      strLst.add(new Integer(1234));
// на этапе Compile/runtime невозможно определить ошибку
      String str = (String)strLst.get(3);
//    компиляция ok, но будет runtime ClassCastException
   }
}
```

*Замечание*. Мы могли бы использовать оператор создания объектов типа класс для проверки правильного типа перед добавлением. Но опять же,

при создании объектов (инстанцировании) проблема обнаруживается во время выполнения. Как насчет проверки типа во время компиляции?

### Использование дженериков в программах

Давайте напишем наш собственный «безопасный тип» ArrayList

Мы проиллюстрируем использование дженериков путем написания нашего собственного типа изменяемого размера массива для хранения определенного типа объектов (аналогично ArrayList).

Начнем с версии MyArrayList без дженериков:

Листинг 20.4 Пример создания динамически размещаемого массива

```
/* Динамически размещаемый массив, котрый содержит большая часть коллекции java.lang.Object — без дженериков.
*/
public class MyArrayList {
   private int size;
// количество элементов – размер коллекции
   private Object[] elements;

   public MyArrayList() {
//конструктор
      elements = new Object[10];
// начальная инициализация емкостью 10 элементов
      size = 0;
   }

   public void add(Object o) {
      if (size < elements.length) {
         elements[size] = o;
      } else {
//выделить массив большего размера и добавить элемент,
      }
    ++size;
     }
```

```
        public Object get(int index) {
            if (index >= size)
                throw                                    new
IndexOutOfBoundsException("Index: " + index + ", Size:
" + size);
            return elements[index];
        }
        public int size() { return size; }
    }
```

В данном примере класс MyArrayList не является безопасным типом. Например, если мы создаем MyArrayList, который предназначен для хранения только String, но допустим в процессе работы с ним добавляется Integer. Что произойдет? Компилятор не сможет обнаружить ошибку. Это связано с тем, что MyArrayList предназначен для хранения объектов Object, и любые классы Java являются производными от Object.

Листинг 20.5 Пример использования класса MyArrayList

```
    public class MyArrayListTest {
        public static void main(String[] args) {
/*Предусмотрено для хранения списка строк, но не
является типобезопасным*/
            MyArrayList strLst = new MyArrayList();
/*добавление элементов строк (типа String) –это
повышающее или расширяющее преобразование (upcasting)
к типу Object*/

            strLst.add("alpha");
            strLst.add("beta");
/*при получении – необходимо явное понижающее
преобразование (downcasting) назад к String*/
            for (int i = 0; i < strLst.size(); ++i) {
                String str = (String)strLst.get(i);
                System.out.println(str);
            }
```

```
/* случайно добавленный не-String объект, произойдет
вызов во время выполнения ClassCastException.
Компилятор не может отловить ошибку*/
          strLst.add(new Integer(1234));
//компиляция/выполнение - не можем обнаружить эту
ошибку
          for (int i = 0; i < strLst.size(); ++i) {
             String str = (String)strLst.get(i);
/*компиляция ok, при выполнении (runtime)
ClassCastException*/
             System.out.println(str);
          }
       }
    }
```

Если вы намереваетесь создать список объектов String, но непреднамеренно добавленный в этот список не-String-объекты будут преобразованы к типу Object. Компилятор не сможет проверить, является ли понижающее преобразование типов действительным во время компиляции (это известно как позднее связывание или динамическое связывание). Неправильное понижающее преобразование типов будет выявлено только во время выполнения программы, в виде исключения ClassCastException, а это слишком поздно, для того чтобы внести изменения в код и исправить работу программы. Компилятор не сможет поймать эту ошибку в момент компиляции. Вопрос в том, как заставить компилятор поймать эту ошибку и обеспечить безопасность использования типа во время выполнения.

### Классы дженерики или параметризованные классы

В JDK введены так называемые обобщенные или параметризованные типы – generics или по-другому обобщенные типы для решения вышеописанной проблемы. Параметризованных (generic) классы и методы, позволяют использовать более гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Использование параметризации позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

Дженерики или обобщенные типы позволяют вам абстрагироваться от использования конкретных типов. Вы можете создать класс с таким

общим типом и предоставить информацию об определенном типе во время создания экземпляра объекта типа класс. А компилятор сможет выполнить необходимую проверку типов во время компиляции. Таким образом, вы сможете убедиться, что во время выполнения программы не возникнет ошибка выбора типа еще на этапе компиляции, что как раз является безопасностью для используемого типа.

Рассмотрим пример:

Листинг 20.6 Объявление интерфейса java.util.List <E>:

```
public interface List<E> extends Collection<E> {
   boolean add(E o);
   void add(int index, E element);
   boolean addAll(Collection<? extends E> c);
   boolean containsAll(Collection<?> c);
   ......
}
```

Такая запись - <E> называется формальным параметром типа, который может использоваться для передачи параметров «типа» во время создания фактического экземпляра типа. Механизм похож на вызов метода Напомним, что в определении метода мы всегда объявляем формальные параметры для передачи данных в метод (при описании метода используются формальные параметры, а при вызове на их место подставляются аргументы). Например как представлено ниже:

```
// Определение метода
public static int max(int a, int b) {
// где int a, int b это формальные параметры
   return (a > b) ? a : b;
}
```

А во время вызова метода формальные параметры заменяются фактическими параметрами (аргументами). Например так:

```
//Вызов: формальные параметры, замененные фактическими параметрами
int maximum = max(55, 66);
// 55 и 66 теперь фактические параметры
int a = 77, b = 88;
maximum = max(a, b);            // a и b теперь фактические параметры
```

Параметры формального типа, используемые в объявлении класса, имеют ту же цель, что и формальные параметры, используемые в объявлении метода. Класс может использовать формальные параметры типа для получения информации о типе, когда экземпляр создается для этого класса. Фактические типы, используемые во время создания, называются фактическими типами параметров.

Вернемся к java. util.List <E>, итак в действительности, когда тип определен, например List <Integer>, все вхождения параметра формального типа <E> заменяются актуальным или фактическим параметром типа <Integer>. Используя эту дополнительную информацию о типе, компилятор может выполнить проверку типа во время компиляции и убедиться, что во время выполнения не будет ошибки при использовании типов.

## Конвенция кода Java об именах для формальных типов

Мы должны помнить, что написания “чистого кода” необходимо руководствоваться конвенцией кода на Java. Поэтому, для создания имен формальных типов используйте один и тот же символ в верхнем регистре. Например,

- <E> для элемента коллекции;
- <T> для обобщенного типа;
- <K, V> ключ и значение.
- <N> для чисел
- S,U,V, и т.д. для второго, третьего, четвертого типа параметра

Рассмотрим пример параметризованного или обобщенного типа как класса. В нашем примере класс GenericBox принимает общий параметр типа E, содержит содержимое типа E. Конструктор, геттер и сеттер работают с параметризованным типом E. Метод нашего класса toString() демонстрирует фактический тип содержимого.

Листинг 20.7 Пример параметризованного класса

```
public class GenericBox<E> {
   // Private переменная класса
   private E content;

   // конструктор
   public GenericBox(E content) {
      this.content = content;
```

```
    }

    public E getContent() {
       return content;
    }

    public void setContent(E content) {
       this.content = content;
    }

    public String toString() {
      return content + " (" + content.getClass() +
")";
    }
}
```

В следующем примере мы создаем GenericBoxes с различными типами (String, Integer и Double). Обратите внимание, что при преобразовании типов происходит автоматическая автоупаковка и распаковка для преобразования между примитивами и объектами-оболочками (мы с вами это изучали ранее).

Листинг 20.8 Пример использования параметризованного класса

```
public class TestGenericBox {
    public static void main(String[] args) {
        GenericBox<String>     box1     =     new
GenericBox<String>("Hello");
        String str = box1.getContent(); // явного
понижающего преобразования (downcasting) не требуется
        System.out.println(box1);
        GenericBox<Integer>     box2     =     new
GenericBox<Integer>(123);
//автоупаковка типа int в тип Integer
        int i = box2.getContent();     // (downcast)
понижающее преобр. к типу Integer, автоупаковка в тип
int
        System.out.println(box2);
```

```
            GenericBox<Double>      box3      =      new
GenericBox<Double>(55.66);
////автоупаковка типа double в тип  Double
            double  d  =  box3.getContent();          //
(downcast) понижающее преобр.  к типу Double,
//автоупаковка в тип double
            System.out.println(box3);
        }
    }
```

Вывод программы:

```
Hello (class java.lang.String)
123 (class java.lang.Integer)
55.66 (class java.lang.Double)
```

**Задания на практическую работу №20**

1. Создать обобщенный класс с тремя параметрами (T, V, K).
2. Класс содержит три переменные типа (T, V, K), конструктор, принимающий на вход параметры типа (T, V, K), методы возвращающие значения трех переменных. Создать метод, выводящий на консоль имена классов для трех переменных класса.
3. Наложить ограничения на параметры типа: T должен реализовать интерфейс Comparable (классы оболочки, String), V должен реализовать интерфейс Serializable и расширять класс Animal, K
4. Написать обобщенный класс MinMax, который содержит методы для нахождения минимального и максимального элемента массива. Массив является переменной класса. Массив должен передаваться в класс через конструктор. Написать класс Калькулятор (необобщенный), который содержит обобщенные статические методы - sum, multiply, divide, subtraction. Параметры этих методов - два числа разного типа, над которыми должна быть произведена операция.
5. Написать класс Matrix, на основе обобщенного типа, реализовать операции с матрицами

**ССЫЛКИ ДЛЯ ЧТЕНИЯ**

1) http://java-s-bubnom.blogspot.com/2015/07/generic.html
2) http://www.quizful.net/post/java-generics-tutorial
3) http://crypto.pp.ua/2010/06/parametrizovannye-klassy-java/

4) https://habr.com/post/207360/
5) https://habr.com/ru/company/sberbank/blog/416413/
6) https://www.geeksforgeeks.org/wildcards-in-java/
7) https://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html
8) https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html