Практическая работа № 21. Стирание типов в Джава

Цель данной практической работы – научиться работать с обобщенными типами в Java и применять прием стирание типов разработке программ на Джава

Теоретические сведения. Стирание типов

Из предыдущего примера может создаться видимость того, что компилятор заменяет параметризованный тип <E> актуальным или фактическим типом (таким как String, Integer) во время создания экземпляра объекта типа класс. Если это так, то компилятору необходимо будет создавать новый класс для каждого актуального или фактического типа (аналогично шаблону C ++).

На самом же деле происходит следующее - компилятор заменяет всю ссылку на параметризованный тип E на ссылку на Object, выполняет проверку типа и вставляет требуемые операторы, обеспечивающие понижающее приведения типов. Например, GenericBox компилируется следующим образом (который совместим с кодами без дженериков):

Листинг 21.1 Пример класса GenericBox с полем Object

```
public class GenericBox {
   // Private переменная
   private Object content;

   // Конструктор
   public GenericBox(Object content) {
      this.content = content;
   }

   public Object getContent() {
      return content;
   }

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

   public String toString() {
```

```
            return content + " (" + content.getClass() +
")";
        }
    }
```

Компилятор сам вставляет требуемый оператор понижения типа (downcasting) в код:

```
    GenericBox box1 = new GenericBox("Hello");   //
upcast безопасно для типов
    String str = (String)box1.getContent();       //
компилятор вставляет операцию понижения типа
(downcast)
    System.out.println(box1);
```

Вывод. Таким образом, для всех типов используется одно и то же определение класса. Самое главное, что байт-код всегда совместим с теми классами, у которых нет дженериков. Этот процесс называется *стиранием типа.*

Рассмотрим операцию “стирания типов” с помощью нашего «безопасного типа» ArrayList, который мы рассматривали ранее в паримере.

Вернемся, к примеру MyArrayList. С использованием дженериков мы можем переписать нашу программу как показано в листинге ниже:

Листинг 21.2 Пример параметризованного класса GenericBox

```
//Динамически выделяемая память для массива с
дженериками
    public class MyGenericArrayList<E> {
       private int size;
    // количество элементов- емкость списка
       private Object[] elements;

       public MyGenericArrayList() {  //конструктор
          elements = new Object[10];
// выделяем память сразу для 10 элементов списка при
его создании
          size = 0;
       }
```

```
        public void add(E e) {
           if (size < elements.length) {
              elements[size] = e;
           } else {
// добавляем элемент к списку и увеличиваем счетчик количества элементов

           }
           ++size;
        }

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

        public int size() { return size; }
     }
```

В объявлении MyGenericArrayList <E> объявляется класс-дженерик с формальным параметром типа <E>. Во время фактического создания объектов типа класс, например, MyGenericArrayList <String>, определенного типа <String> или параметра фактического типа, подставляется вместо параметра формального типа <E>.

Помните! Что Дженерики реализуются компилятором Java в качестве интерфейсного преобразования, называемого стиранием, которое переводит или перезаписывает код, который использует дженерики в не обобщенный код (для обеспечения обратной совместимости). Это преобразование стирает всю информацию об общем типа. Например, ArrayList <Integer> станет ArrayList. Параметр формального типа, такой как <E>, заменяется объектом по умолчанию (или верхней границей типа). Когда результирующий код не корректен, компилятор вставляет оператор преобразования типа.

Следовательно, код, транслированный компилятором, т о есть переведенный код выглядит следующим образом:

Листинг 21.3 Пример кода транслированногоо компилятором

```
public class  MyGenericArrayList {
   private int size;     // количество элементов
   private Object[] elements;

   public MyGenericArrayList() {  // конструктор
      elements = new Object[10];   // выделяем
память для первых 10 объектов
      size = 0;
   }

   // Компилятор заменяет параметризованный тип E
на Object, но проверяет, что параметр e имеет тип E,
когда //он используется для обеспечения безопасности
типа

   public void add(Object e) {
      if (size < elements.length) {
         elements[size] = e;
      } else {
         // allocate a larger array and add the
element, omitted
      }
      ++size;
   }

   // Компилятор заменяет E на Object и вводит
оператор понижающего преобразования типов (E <E>) для
//типа возвращаемого значения при вызове метода

   public Object get(int index) {
      if (index >= size)
```

```
            throw                                    new
IndexOutOfBoundsException("Index: " + index + ", Size:
" + size);
            return (Object)elements[index];
        }

        public int size() {
            return size;
     }
    }
```

Когда класс создается с использованием актуального или фактического параметра типа, например. MyGenericArrayList <String>, компилятор гарантирует, что add(E e) работает только с типом String. Он также вставляет соответствующий оператор понижающее преобразование типов в соответствие с типом возвращаемого значения E для метода get(). Например,

Листинг 21.4 Пример использования класса дженерика

```
    public class MyGenericArrayListTest {
        public static void main(String[] args) {
            // безопасный тип для хранения списка
объектов Strings (строк)
            MyGenericArrayList<String>   strLst  =  new
MyGenericArrayList <String>();

            strLst.add("alpha");    // здесь компилятор
проверяет, является ли аргумент типом String
            strLst.add("beta");

            for (int i = 0; i < strLst.size(); ++i) {
                String str = strLst.get(i); //компилятор
вставляет оператор
    //понижающего преобразования operator (String)
                System.out.println(str);
            }
            strLst.add(new    Integer(1234));        //
компилятор обнаруживает аргумент,
```

```
    //который не является объектом String, происходит ошибка компиляции
    }
    }
```

Выводы: с помощью дженериков компилятор может выполнять проверку типов во время компиляции и обеспечивать безопасность использования типов во время выполнения.

Запомните. В отличие от «шаблона» в C ++, который создает новый тип для каждого определенного параметризованного типа, в Java класс generics компилируется только один раз, и есть только один файл класса, который используется для создания экземпляров для всех конкретных типов.

Обобщенные методы (параметризованные методы)

Методы также могут быть определены с помощью общих типов (аналогично родовому классу). Например,

Листинг 21.5 Пример параметризованного метода

```
    public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {
        for (E e : a) lst.add(e);
    }
```

При объявлении обобщенного метода или метода-дженерика могут объявляться параметры формального типа (например, такие как <E>, <K, V>) перед возвращаемым типом (в примере выше это static <E> void). Параметры формального типа могут затем использоваться в качестве заполнителей для типа возвращаемого значения, параметров метода и локальных переменных в общем методе для правильной проверки типов компилятором.

Подобно классам-дженерикам, когда компилятор переводит общий метод, он заменяет параметры формального типа, используя операцию стирания типа. Все типы заменяются типом Object по умолчанию (или верхней границей типа – типом классом, стоящим на самой вершине иерархии наследования). Переведенная на язык компилятора версия программы выглядит следующим образом:

Листинг 21.6 Пример трансляции кода компилятором

```
    public static void ArrayToArrayList(Object[] a,
ArrayList lst) {
    // компилятор проверяет, есть ли тип E[],
    // lst имеет тип ArrayList<E>
     ArrayList<E> for (Object e : a) lst.add(e);
    // компилятор проверяет является ли e типом E
    }
```

Компилятор всегда проверяет, что a имеет тип E[], lst имеет тип ArrayList <E>, а e имеет тип E во время вызова для обеспечения безопасности типа. Например,

Листинг 21.7 Пример класса с использованием параметризованного метода

```
    import java.util.*;
    public class TestGenericMethod {
       public static <E> void ArrayToArrayList(E[] a,
ArrayList<E> lst) {
          for (E e : a) lst.add(e);
       }
    public static void main(String[] args) {
    ArrayList<Integer> lst = new ArrayList<Integer>();
          Integer[] intArray = {55, 66};     //
автоупаковка
          ArrayToArrayList(intArray, lst);
          for (Integer i : lst) System.out.println(i);

          String[] strArray = {"one", "two", "three"};
          //ArrayToArrayList(strArray, lst);
    //ошибка компиляции ниже
       }
TestGenericMethod.java:16:<E>ArrayToArrayList(E[],jav
a.util.ArrayList<E>) in TestGenericMethod cannot be
applied                                          to
(java.lang.String[],java.util.ArrayList<java.lang.Int
eger>) ArrayToArrayList(strArray, lst);
    }
```

У дженериков есть необязательный синтаксис для указания типа для общего метода. Вы можете поместить фактический тип в угловые скобки <> между оператором точки и именем метода.

```
TestGenericMethod.<Integer>ArrayToArrayList(intArray, lst);
```

Обратите внимание на точку после имени метода

Подстановочный синтаксис в Java (WILD CARD)

Одним из наиболее сложных аспектов generic-типов (обобщенных типов) в языке Java являются wildcards (подстановочные символы, в данном случае – «?»), и особенно – толкование и разбор запутанных сообщений об ошибках, происходящих при wildcard capture (подстановке вычисляемого компилятором типа вместо wildcard). В своем труде «Теория и практика Java» (Java theory and practice) старейший Java-разработчик Брайен Гетц расшифровывает некоторые из наиболее загадочно выглядящих сообщений об ошибках, выдаваемых компилятором «javac», и предлагает решения и варианты обхода, которые помогут упростить использование generic-типов.

Рассмотрим следующую строку кода:

```
ArrayList<Object> lst = new ArrayList<String>();
```

Компиляция вызовет ошибку - «несовместимые типы», поскольку ArrayList <String> не является ArrayList <Object>. Эта ошибка противоречит нашей интуиции в отношении полиморфизма, поскольку мы часто присваиваем экземпляр подкласса ссылке на суперкласс.

Рассмотрим эти два утверждения:

```
List<String> strLst = new ArrayList<String>();
// строчка 1
List<Object> objLst = strList;
// строчка 2 – ошибка компиляции
```

Выполнение строки 2 генерирует ошибку компиляции. Но если строка 2 выполняется, то некоторые объекты добавляются в objLst, а strLst будут «повреждены» и больше не будет содержать только строки. (так-как переменные objLst и strLst содержат одинаковую ссылку или ссылаются на одну и ту-же область памяти).

Учитывая вышеизложенное, предположим, что мы хотим написать метод printList (List <.>) Для печати элементов списка. Если мы определяем

метод как printList (List <Object> lst), он может принимать только аргумент List <object>, но не List <String> или List <Integer>. Например,

Листинг 21.8 Пример параметризованного метода для печати элементов списка

```
import java.util.*;
public class TestGenericWildcard {

   public static void printList(List<Object> lst) {   // принимает только список объектов,не список подклассов объектов
      for (Object o : lst) System.out.println(o);
   }

   public static void main(String[] args) {
      List<Object>       objLst       =       new ArrayList<Object>();
      objLst.add(new Integer(55));
      printList(objLst);   // соответствие

      List<String>       strLst       =       new ArrayList<String>();
      strLst.add("one");
      printList(strLst);  // ошибка компиляции
   }
}
```

Использование подстановочного знака без ограничений в описании типа <?>

Чтобы разрешить проблему, описанную выше, необходимо использовать подстановочный знак (?), он используется в дженериках для обозначения любого неизвестного типа. Например, мы можем переписать наш printList () следующим образом, чтобы можно было передавать список любого неизвестного типа.

```
public static void printList(List<?> lst) {
  for (Object o : lst) System.out.println(o);
}
```

Использование подстановочного знака в начале записи типа <? extends тип>

Подстановочный знак <? extends type> обозначает тип и его подтип. Например,

```
public static void printList(List<? extends Number> lst) {
  for (Object o : lst) System.out.println(o);
}
```

List<? extends Number> принимает список Number и любой подтипNumber, например, List <Integer> и List <Double>. Понятно, что обозначение типа <?> можно интерпретировать как <? extends Object>, который применим ко всем классам Java.

Другой пример,

```
//List<Number> lst = new ArrayList<Integer>(); // ошибка компиляции
List<? extends Number> lst = new ArrayList<Integer>();
```

На самом деле стирание типов обеспечивает совместимость вашего кода со старыми версиями java, которые могут вообще не содержать дженериков.

Задания на практическую работу №21

1. Написать метод для конвертации массива строк/чисел в список.
2. Написать класс, который умеет хранить в себе массив любых типов данных (int, long etc.).
3. Реализовать метод, который возвращает любой элемент массива по индексу.
4. Написать функцию, которая сохранит содержимое каталога в список и выведет первые 5 элементов на экран.
5. *Реализуйте вспомогательные методы в классе Solution, которые должны создавать соответствующую коллекцию и помещать туда переданные объекты. Методы newArrayList, newHashSet параметризируйте общим типом T. Метод newHashMap параметризируйте парой <K, V>, то есть типами K- ключ и V-значение. Аргументы метода newHashMap

должны принимать. Класс содержит три переменные типа (T, V, K), конструктор, принимающий на вход