Мультимедийный курс

Программирование на Java

Лекция 6.1

Параметризированные типы (Обобщения, Generics)

Параметризированные (generic) типы - классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр (***параметр типа***) .

Параметризированные типы, позволяют использовать более гибкую и в то же время достаточно строгую типизацию, обеспечивая безопасность типов

1. *Ограничение типа, применение метасимволов*
2. *Настраиваемые методы, интерфейсы иерархии классов*
3. *Параметры типа, параметризированные классы*
4. *Реализация настраиваемых типов, ограничения по применению*

Классы, использующие параметр типа, являются настраиваемыми классами или параметризованными типами

Применение параметризированного класса:

```
Gen<Integer>  iob = new  Gen<Integer>(88) ;
```

Полная идентичность аргументов типа

Аргумент типа, задаваемый при объявлении параметризированного типа – может быть только классом, а не примитивным типом !!!

Разные аргументы:

```
Gen<Integer> iob = new Gen<Integer>(88) ;
Gen<String> strOb = new Gen <String> ("Generics Test");
iob = strOb;          // Неверно - разные типы !!!
Gen<String> s =
        new Gen<Integer>(777) ; ./* Неверно - разные типы !!!
 т.е. ссылка на одну конкретную версию обобщенного
типа несовместима с другой версией того же самого
обобщенного типа. */
```

Пример1: Параметризированный класс с одним параметром типа:

```
package chapt03;

public class Optional <T> {
        private T value;

        public Optional() {
        }
        public Optional(T value) {
                this.value = value;
        }
        public T getValue() {
                return value;
        }
        public void setValue(T val) {
                value = val;
        }
        public String toString() {
                if (value == null) return null;
            return value.getClass().getName() + " " + value;
        }
}
```

Пример1: Параметризированный класс с одним параметром типа:

```
package chapt03;

public class Runner {
        public static void main(String[] args) {
//параметризация типом Integer
Optional<Integer> ob1 =
new Optional<Integer>();
                ob1.setValue(1);
//ob1.setValue("2");// ошибка компиляции: недопустимый тип
                int v1 = ob1.getValue();
                System.out.println(v1);
//параметризация типом String
                Optional<String> ob2 =
new Optional<String>("Java");
                String v2 = ob2.getValue();
                System.out.println(v2);
        //ob1 = ob2; //ошибка компиляции – параметризация не ковариантна
```

Пример1: Параметризированный класс с одним параметром типа:

```
                    //параметризация по умолчанию – Object
            Optional ob3 = new Optional();
            System.out.println(ob3.getValue());
            ob3.setValue("Java SE 6");
            System.out.println(ob3.toString());/* выводится тип объекта, а не тип
                                                                параметризации */

            ob3.setValue(71);
            System.out.println(ob3.toString());

            ob3.setValue(null);
        }
}
```

В результате выполнения этой программы будет выведено:

```
1
Java
null
java.lang.String Java SE 6
java.lang.Integer 71
```

Пример1: Параметризированный класс с одним параметром типа:

В следующей строке кода: class Gen<T> { ,

где T обозначает имя параметра типа.

Это имя используется в качестве заполнителя, вместо которого в дальнейшем подставляется имя конкретного типа, передаваемого классу Gen при создании объекта.

Обозначение T применяется в классе Gen всякий раз, когда требуется параметр типа.

Всякий раз. когда объявляется параметр типа, он указывается в угловых скобках < >.

Параметризированный класс с несколькими параметрами типа:

```
// Два параметра типа задаются списком через запятую
class TwoGen<T,  V>  {
    T ob1;
    V ob2;
    TwoGen(T o1, V o2)  {  // конструктор
     ob1 = o1;
     ob2 = o2;
  }

// Использование класса

TwoGen <Integer, String> tgObn =
          new TwoGen <Integer, String> (88, "Generics");
```

Ограничения на типы, передаваемые параметру типа:

- В качестве верхней границы задается суперкласс, от которого должны быть унаследованы все аргументы типа:

  Gen <T **extends** *superclass>*

  Gen <T **extends** *Number>*

  Аргументы параметра типа в этом случае:

  ✔ только тип ***Number*** или его подклассы Integer, Float

Применение метасимвольных аргументов

необходимо в случае, если параметр типа невозможно определить

Метасимвол: **?**

Пример

```
class  Stats<T> {
   T[]  nums;
    Stats(T[] o) {
    nums  = o;
    }
// Вычисляет среднее арифметич. элементов массива
    double average() {
        double sum = 0.0;
        for (int i=0;  i < nums.length; i++)
           sum += nums[i].doubleValue(); //  метод кл. Number – приводит к
                                         //  типу  Double
           return sum / nums.length;
    }
}
```

Пример (продолжение)

```
boolean sameAvg(Stats<?> ob)  {        // любой объект типа Stats,
// если T, то ожидается тип передаваемый вызывающим объектом
  if  (average() == ob.average())
     return true;
  else
     return false;
}
. . .
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double>  dob = new Stats<Double>(dnums);

System.out.print("Средние арифметические iob и dob ");
   if (iob. sameAvg (dob) ) // вызывающий об. и параметр  –
                            // объекты одного класса
      System.out.println("совпадают.");
   else
      System, out.println("отличаются.");
. . .
```

Ограничение метасимвольных аргументов

Задание верхней границы:

<? extends *superclass*>

superclass - имя класса, который служит верхней границей

Задание нижней границы:

<? super *subclass*>

Допустимые аргументы - суперклассы класса ***subclass.*** ***subclass*** не является допустимым типом аргумента

Пример

```
static void showXYZ(Coords<? extends ThreeD>  c) {
    System.out.println("X Y Z Coordinates:");
    for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords [i].x + " " +
                         c.coords[i].y + " " +
                         c.coords[i].z);
    System.out.println();
  }
```

Параметризированные методы

- могут иметь один или несколько собственных параметров типа
- могут создаваться и внутри непараметризированного класса
- могут быть как статическими, так и нестатическими

Синтаксис записи

<список_парам._типа> возвр._знач.

имя_метода(список_парам.) { ...}

Пример:

```
static <T, V extends T> boolean isIn(T x, V[] y) {
```

Пример:

```
// Параметризированный метод
    static <T, V extends T> boolean isIn(T x, V[]  y)  {
        for(int i=0; i < y.length; i++)
            if(x.equals(y[i])) return true;
        return false;
      }
// Вызов метода
    public static void main(String args[]) {
        Integer nums[] = { 1, 2, 3, 4, 5 };
        if(isIn(2, nums))
           System.out.println("2 содержится в массиве");
    }
```

Пример:

```
class GenCons {
  private double val;
  <T extends Number> GenCons(T arg) {  // Параметризированный только метод
    val = arg.doubleValue();
  }

  void showval() {
    System.out.println("val: " + val);
  }
}

class GenConsDemo {
  public static void main(String args[]) {
    GenCons test = new GenCons(100);
    GenCons test2 = new GenCons(123.5F);
    test.showval();
    test2.showval();
  }
}
```

Вывод программы:

```
val: 100,0
val:  123,5
```

Настраиваемые интерфейсы

- задаются так же, как настраиваемые классы

```
interface MinMax<T extends Comparable<T>> {
```

- класс, реализующий интерфейс

Та же граница

```
Class MyClass<T extends Comparable<T>>
      implements MinMax<T> {
```

Не повторяется

Иерархии параметризированных классов

- параметризированный класс может быть суперклассом или быть подклассом
- аргументы типа, необходимые суперклассу, должны передаваться всем подклассам !!!

```
class Gen<T> {
  T ob;
Gen(T o) {
  ob =  o;
}
T getob() {
    return ob;
  }
}
```

```
class Gen2<T> extends Gen<T>{
Gen2(T o) {
    super(o); //  передача суперклассу
}
}
…
Gen2<Integer> x = new  Gen2<Integer>(108);
System.out.print(x.getob());
```

Иерархии настраиваемых классов (продолжение)

- в подклассе всегда определяются параметры типа, требующиеся для его настраиваемого суперкласса
- подкласс может иметь собственные параметры типа

```
class Gen2<T, V> extends Gen<T>  {
   V ob2;
   Gen2(T  o, V o2) {
      super(o); // передача аргумента типа
        //  конструктору супертипа
      ob2 = o2;
   }
}
```

Применение настраиваемых типов в коллекциях

- все классы и интерфейсы, связанные с классами ***ArrayList***, ***LinkedList*** и ***TreeSet*** - параметризированные:

```
ArrayList<String> list = new ArrayList<String>();
```

- преимущества:
  - гарантируется сохранение в коллекции ссылок на объекты только нужного типа
  - исключается необходимость явного приведения типа ссылки, при извлечении из коллекции

Сравнение типов настраиваемой иерархии:

операция

объект **instanceof** *тип*

возвращает *true,* если *объект* имеет заданный *тип* или м. б. преобразован к нему

```
if (iOb instanceof Gen2<?>)
   System.out.println("iOb  совместим с Gen2");
```

Приведение типов: преобразование одного экземпляра параметризированного класса в другой

```
(Gen<Integer>)  iob2
```

если они взаимно совместимы

Реализация в Java обобщенных типов

При **компиляции**:

- информация о обобщенных типах **удаляется**

  (эффект стирания)

- **параметры типа заменяются ограничивающими их типами** (если заданы) либо *Object*

- все параметризованные классы используют один класс:

```
Gen<Integer> iOb  = new Gen<Integer>(99);
   Gen<Float>  fOb =  new Gen<Float>(102.2F);
   Systern.out.println(iOb.getclass () .getName()) ;
   System, out.println(fOb.getClass (). getName());
```

Результат:

Gen

Gen

Поэтому нельзя: *запрашивать тип в процессе выполнения программы*

```
public class MyList<E> {
  public E[] toArray() {
    return new E[5]; // compile error
  }
  public boolean canAdd(Object o) {
    return (o instanceof E); // compile error –}
  public E convertToE(Object o) {
    return (E) o; // compile warning, unchecked cast
  }
}
```

Raw Type

- Можно создать объект настраиваемого (генефицированного) класса без указания типов аргументов
- Классы Pre-J2SE 5.0 продолжают функционировать под J2SE 5.0 JVM как raw тип

```
// ??????
List<String> ls = new LinkedList<String>();
// Raw type
List lraw = new LinkedList();
```

Ограничения

T t = new T(); // конструктор ?

✔ Статические члены класса не могут использовать параметры типа

static T t; //??????

Ограничения (продолжение)

Создание экземпляров универсальных типов

```
class Test<T> {
   T values;                  //ok
   Test(T[] n) {
      values = new T[10];     // базовый тип - параметр типа

      values = n;             //ok
    }
}
```

Массивы

```
Test<Integer> iTest[] = new Test<Integer>[10]          //
 нельзя объявлять с аргументами типа
Test<?> iTest[] = new Test<?>[10]                      //ok
```

Ограничения (продолжение)

✔ Исключения

Невозможно

- сгенерировать или перехватить исключение, описываемое универсальным объектом
- создать параметризированный класс, расширяющий класс **Throwable**
- использовать параметр типа в выражении **catch**

Допустимо использовать параметр типа в выражении **throws**