Конспект лекции
Родовые типы Java (Generics)

Цель и задачи лекции

Цель – изучить использование родовых типов Generics в Java.

Задачи:

1. Дать понятие параметризированных классов
2. Изучить применение Generics и ограничений с их помощью
3. Дать определение ковариантности и инвариантности

План занятия

1. Параметризованные классы
2. Diamond оператор
3. Универсальные методы (Generic methods)
4. Ограничения на допустимые типы
5. Ковариантность, контравариантность и инвариантность
6. Wildcards

Параметризованные классы

К наиболее важным новшествам версии языка J2SE 5 можно отнести появление параметризации (generic) классов и методов, позволяющей использовать гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Применение generic-классов для создания типизированных коллекций будет рассмотрено в главе «Коллекции». Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

Generic класс объявляется следующим способом:

```
public class ClassName<T1, T2, ...> {
    // ...
}
```

Ниже приведен пример generic-класса с параметрами:

```
class Pair<T1, T2> {
    T1 object1;
    T2 object2;

    Pair(T1 one, T2 two) {
```

```
        object1 = one;
        object2 = two;
    }

    public T1 getFirst() {
        return object1;
    }

    public T2 getSecond() {
        return object2;
    }
}

class Test {
    public static void main(String[] args) {
        Pair<Integer, String> pair = new Pair<Integer, String>(6, "Apr");
        System.out.println(pair.getFirst() + pair.getSecond());
    }
}
```

Diamond оператор

Чтобы упростить жизнь программистам в Java 7 был введён алмазный синтаксис (diamond syntax), в котором можно опустить параметры типа. Т.е. можно предоставить компилятору определение типов при создании объекта. Вид упрощённого объявления:

```
Pair<Integer, String> pair = new Pair<>(6, "Apr");
```

Следует обратить внимание, что возможны ошибки связанные с отсутствием "<>" при использовании алмазного синтаксиса:

```
Pair<Integer, String> pair = new Pair(6, "Apr");
```

В случае с примером кода выше мы просто получим предупреждение от компилятора, Поскольку Pair является дженерик-типом и были забыты "<>" или явное задание параметров, компилятор рассматривает его в качестве простого типа (raw type) с Pair принимающим два параметра типа объекта. Хотя такое поведение не вызывает никаких проблем в данном сегменте кода, это может привести к ошибке. Здесь необходимо пояснение понятия простого типа.

Посмотрим на первый фрагмент кода:

```
List list = new LinkedList();
list.add("First");
list.add("Second");
List<String> list2 = list;
for (Iterator<String> itemItr = list2.iterator(); itemItr.hasNext(); )
    System.out.println( itemItr.next() );
```

и аналогичный второй фрагмент кода:

```
List list = new LinkedList<>();
list.add("First");
list.add("Second");
List<String> list2 = list;
for (Iterator<String> itemItr = list2.iterator(); itemItr.hasNext(); )
    System.out.println( itemItr.next() );
```

По результатам выполнения оба фрагмента аналогичны, но у них разная идея. В первом случае мы имеем место с простым типом, во вторым – с дженериком. Теперь сломаем это дело – заменим в обоих случаях

```
list.add("Second");
```
на

```
list.add(10);
```

Для простого типа получим ошибку времени выполнения (java.lang.ClassCastException), а для второго – ошибку компиляции. При использовании простых типов, вы теряете преимущество безопасности типов, предоставляемое дженериками.

Универсальные методы (Generic methods)

По аналогии с универсальными классами (дженерик-классами), можно создавать универсальные методы (дженерик-методы), то есть методы, которые принимают общие типы параметров. Универсальные методы не надо путать с методами в дженерик-классе. Универсальные методы удобны, когда одна и та же функциональность должна применяться к различным типам. Например, есть многочисленные общие методы в классе java.util.Collections.

Параметризованный (generic) метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра, и может быть записан, например, в виде:

```
<T extends Тип> returnType methodName(T arg) { }
<T> T[ ] methodName(int count, T arg) { }
```

Описание типа должно находиться перед возвращаемым типом. Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после extends. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

Generic-методы могут находиться как в параметризованных классах, так и в обычных. Параметр метода может не иметь никакого отношения к параметру своего generic-класса. Причем такому методу разрешено быть статическим, так как параметризацию обеспечивает сам метод, а не класс, в котором он объявлен. Метасимволы применимы и к generic-методам.

```
public class SimpleActionCourse {
    public <T1 extends Course> SimpleActionCourse(T1 course) { // конструктор
        // реализация
    }
    public <T2> SimpleActionCourse() { // конструктор
        // реализация
    }
    public <T3 extends Course> float calculateMark(T3 course) {
        // реализация
    }
    public <T4> boolean printReport(T4 course) {
        // реализация
    }
    public <T5> void check() {
        // реализация
```

```
    }
}
```

Создание экземпляра с параметром и вызов параметризованного метода с параметром выглядят следующим образом:

```
SimpleActionCourse sap = new SimpleActionCourse(new Course());
sap.printReport(new Course(7112));
```

Создание экземпляра с использованием параметризованного конструктора без

параметров требует указания типа параметра перед именем конструктора

```
SimpleActionCourse sa = new <String>SimpleActionCourse();
```

Аналогично для метода без параметров

```
sa.<Integer>check();
```

Ограничения на допустимые типы

Java предоставляет возможность ограничить допустимые generic-типы. При объявлении параметризованного класса или метода можно использовать конструкции вида <T extends SomeClass>. В этом случае качестве T может подставить только Phone и его наследников. Можно указать один класс и несколько интерфейсов, разделяя их оператором &.

Кроме того, механизм Generic предоставляет возможность оперировать с шаблонами (wildcard) — вместо типа использовать вопросительный знак. Они используются тогда, когда нужно абстрагироваться от конкретных аргументов типа, и позволяют использовать его там, где не требуется знать конкретные типы параметров. Использовать можно маски без ограничений (например, Map<key, ?>, а также с ограничениями List<? extends MyClass>, List<? super MyClass>).

Ковариантность, контравариантность и инвариантность

Ковариантность — это сохранение иерархии наследования исходных типов в производных типах в том же порядке. Например, если Кошка — это подтип Животные, то Множество<Кошки> — это подтип Множество<Животные>. Следовательно, с учетом принципа подстановки можно выполнить такое присваивание:

Множество<Животные> = Множество<Кошки>

Контравариантность — это обращение иерархии исходных типов на противоположную в производных типах. Например, если Кошка — это подтип Животные, то Множество<Животные> — это подтип Множество<Кошки>. Следовательно, с учетом принципа подстановки можно выполнить такое присваивание:

Множество<Кошки> = Множество<Животные>

Инвариантность — отсутствие наследования между производными типами. Если Кошка — это подтип Животные, то Множество<Кошки> не является подтипом

Множество<Животные> и Множество<Животные> не является подтипом Множество<Кошки>.

Массивы в Java ковариантны. Тип S[] является подтипом T[], если S — подтип T. Пример присваивания:

```
String[] strings = new String[] {"a", "b", "c"};
Object[] arr = strings;
```

Мы присвоили ссылку на массив строк переменной arr, тип которой - «массив объектов». Если бы массивы не были ковариантными, нам бы это сделать не удалось. Java позволяет это сделать, программа скомпилируется и выполнится без ошибок.

```
arr[0] = 42; // ArrayStoreException. Проблема на этапе выполнения программы
```

Но если мы попытаемся изменить содержимое массива через переменную arr и запишем туда число 42, то получим ArrayStoreException на этапе выполнения программы, поскольку 42 является не строкой, а числом. В этом заключается недостаток ковариантности массивов Java: нет возможности выполнить проверки на этапе компиляции, и может произойти ошибка на этапе выполнения.

Массивы в языке программирования Java являются ковариантными - это означает, что если Integer расширяет Number (как и есть на самом деле), то не только Integer является Number, но и Integer[] тоже является Number[], и вы можете передавать или присваивать Integer[] при вызове Number[]. (Более формально, если Number является супертипом Integer, то Number[] является супертипом Integer[].) Вы можете подумать, что это верно и в отношении родовых типов - то есть List<Number> является супертипом List<Integer>, и что можно передавать List<Integer>, когда нужен List<Number>.

Существует веская причина, чтобы все работало именно так: описанная ситуация привела бы к нарушению независимости родовых типов, которую мы стремимся обеспечить. Представьте, например, что вы могли бы присвоить List<Integer> типу List<Number>. Тогда в следующем фрагменте кода вы смогли бы добавить в List<Integer> элемент, не являющийся Integer:

```
List<Integer> li = new ArrayList<Integer>();
List<Number> ln = li; // не верно
ln.add(new Float(3.1415));
```

Поскольку ln имеет тип List<Number>, добавление Float к нему выглядит совершенно легальным. Но если ln присвоить li, то это нарушило бы независимость от типа, который явно указан при определении li (что li - это список целых чисел), вот почему родовые (generic) типы не могут быть ковариантными.

Дженерики инвариантны. Рассмотрим пример:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<Number> nums = ints; // compile-time error
nums.set(2, 3.14);
assert ints.toString().equals("[1, 2, 3.14]");
```

Если взять список целых чисел, то он не будет являться ни подтипом типа Number, ни каким-либо другим подтипом. Он является только подтипом самого себя. То есть List <Integer> — это List<Integer> и ничего больше. Компилятор позаботится о том, чтобы переменная ints, объявленная как список объектов класса Integer, содержала только объекты класса Integer и ничего кроме них. Проверка производится на этапе компиляции.

Еще одним следствием того, что массивы являются ковариантными, а родовые типы нет, является невозможность создать экземпляр массива родового типа (new List<String>[3] записать нельзя), если типом аргумента не является групповой символ (new List<?>[3] записать можно). Рассмотрим пример:

```
List<String>[] lsa = new List<String>[10]; // не верно
Object[] oa = lsa;  // OK поскольку List<String> является подтипом Object
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[0] = li;
Strings = lsa[0].get(0);
```

Последняя строка вызовет исключительную ситуацию ClassCastException, поскольку вы попытались вставить List<Integer> в List<String>. Поскольку ковариантность массива могла бы позволить вам нарушить независимость родового типа, создание экземпляра массивов родовых типов (за исключением типов, аргументы которых являются неограниченными групповыми символами) была запрещена.

Wildcards

Этот термин в разных источниках переводится по-разному: метасимвольные аргументы, подстановочные символы, групповые символы, шаблоны, маски и т.д.

Generics не всегда инварианты. Рассмотрим примеры:

```
List<Integer> ints = new ArrayList<Integer>();
List<? extends Number> nums = ints;
```

Это ковариантность. List<Integer> — подтип List<? extends Number>

```
List<Number> nums = new ArrayList<Number>();
List<? super Integer> ints = nums;
```

Это контравариантность. List<Number> является подтипом List<? super Integer>.

Запись вида "? extends ..." или "? super ..." — называется wildcard или символом подстановки, с верхней границей (extends) или с нижней границей (super). List<? extends Number> может содержать объекты, класс которых является Number или наследуется от Number. List<? super Number> может содержать объекты, класс которых Number или у которых Number является наследником (супертип от Number).

Литература и ссылки

1. Спецификация языка Java https://docs.oracle.com/javase/specs/jls/se8/html/index.html

2. http://www.ibm.com/developerworks/ru/java/library/j-jtp01255/index.html
3. https://docs.oracle.com/javase/tutorial/extra/generics/
4. https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html
5. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
6. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.

Вопросы для самоконтроля

1. Проанализируйте, скомпилируется ли приведенный ниже код и в какой строке произойдет ошибка

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(3.14);
```

   Если контейнер объявлен с wildcard ? extends, то можно только читать значения. В список нельзя ничего добавить, кроме null. Для того чтобы добавить объект в список нам нужен другой тип wildcard — ? super

2. Почему нельзя получить элемент из списка ниже?

```
public static <T> T getFirst(List<? super T> list) {
    return list.get(0); // compile-time error
}
```

   Нельзя прочитать элемент из контейнера с wildcard ? super, кроме объекта класса Object - <T> Object getFirst(…)

3. Для чего введены Generics?
4. В чем отличие Generics в методах и в классах?
5. Что такое Diamond-оператор?
6. Что такое WildCards?
7. Какой тип данных виден для Generics в Runtime?