7. Generics

Обобщённое программирование — это такой подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания. В Java, начиная с версии J2SE 5.0, добавлены средства обобщённого программирования, синтаксически основанные на C++. Ниже будут рассматриваться **generics (дженерики)** или **<<контейнеры типа T>>** — подмножество обобщённого программирования.

Допустим мы ничего не знаем о дженериках и нам необходимо реализовать специфический вывод на консоль информации об объектах различного типа (с использованием фигурных скобок).

Ниже пример реализации:

```
package test;

class BoxPrinter {
    private Object val;

    public BoxPrinter(Object arg) {
        val = arg;
    }

    public String toString() {
        return "{" + val + "}";
    }

    public Object getValue() {
        return val;
    }
}

class Test {
    public static void main(String[] args) {
        BoxPrinter value1 = new BoxPrinter(new Integer(10));
        System.out.println(value1);

        Integer intValue1 = (Integer) value1.getValue();
        BoxPrinter value2 = new BoxPrinter("Hello world");

        System.out.println(value2);

        // Здесь программист допустил ошибку, присваивая
        // переменной типа Integer значение типа String.
        Integer intValue2 = (Integer) value2.getValue();
    }
}
```

В вышеприведённом коде была допущена ошибка, из-за которой на консоли мы увидим следующее:

```
{10}
{Hello world}
Exception in thread "main" java.lang.ClassCastException: java.lang.String
incompatible with java.lang.Integer
    at test.Test.main(Test.java:29)
```

Теперь на время забудем об этом примере и попробуем реализовать тот же функционал с использованием ~дженериков~ (и повторим ту же ошибку):

```
package test;

class BoxPrinter<T> {
    private T val;

    public BoxPrinter(T arg) {
        val = arg;
    }

    public String toString() {
        return "{" + val + "}";
    }

    public T getValue() {
        return val;
    }
}

class Test {
    public static void main(String[] args) {
        BoxPrinter<Integer> value1 = new BoxPrinter<Integer>(new
Integer(10));

        System.out.println(value1);

        Integer intValue1 = value1.getValue();
        BoxPrinter<String> value2 = new BoxPrinter<String>("Hello world");

        System.out.println(value2);

        // Здесь повторяется ошибка предыдущего фрагмента кода
        Integer intValue2 = value2.getValue();
    }
}
```

Самое существенное отличие (для меня) в том, что при ошибке, аналогичной предыдущей, проблемный код не скомпилируется:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    Type mismatch: cannot convert from String to Integer

    at test.Test.main(Test.java:28)
```

Думаю, многие согласятся, что ошибка компиляции «лучше» ошибки времени выполнения, т.к. чисто теоретически скомпилированный код с ошибкой может попасть туда, куда ему лучше бы и не попадать. Это очевидное достоинство дженериков. Теперь подробнее рассмотрим конструкции, относящиеся к дженерикам в этом примере. Для того, чтобы код скомпилировался, достаточно заменить строку

```
    Integer intValue2 = value2.getValue();
```

на

```
   String stringValue = value2.getValue();
```

Посмотрим на декларацию `BoxPrinter`:

```
    class BoxPrinter<T>
```

После имени класса в угловых скобках `<` и `>` указано имя типа `T`, которое может использоваться внутри класса. Фактически `T` – это тип, который должен быть определён позже (при создании объекта класса).

Внутри класса первое использование `T` в объявлении поля:

```
    private T val;
```

Здесь объявляется переменная дженерик-типа (~generic type~), таким образом её тип будет указан позже, при создании объекта класса `BoxPrinter`.

В `main()`-методе происходит следующее объявление:

```
    BoxPrinter<Integer>  value1
```

Здесь указывается, что `T` имеет тип `Integer`. Грубо говоря, для объекта `value1` все поля `T`-типа его класса `BoxPrinter` становятся полями типа `Integer` (`private Integer val;`).
Ещё одно место, где используется `T`:

```
public BoxPrinter(T arg) {
    val = arg;
}
```

Как и в декларации `val` с типом `T`, вы говорите, что аргумент для конструктора `BoxPrinter` имеет тип `T`. Позже в `main()`-методе, когда будет вызван конструктор в `new`, указывается, что `T` имеет тип `Integer`:

```
    new BoxPrinter<Integer>(new Integer(10));
```

Теперь, внутри конструктора `BoxPrinter`, `arg` и `val` должны быть одного типа, так как оба имеют тип `T`. Например следующее изменение конструктора:

```
    new BoxPrinter<String>(new Integer(10));
```

приведёт к ошибке компиляции.

Последнее место использования `T` в классе – метод `getValue()`:

```
public T getValue() {
    return val;
}
```

Тут вроде тоже всё ясно – этот метод для соответствующего объекта будет возвращать значение того типа, который будет задан при его (объекта) создании.

При создании дженерик-классов мы не ограничены одним лишь типом `T` – их может быть несколько:

```
package test;

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());
    }
}
```

Нет ограничений и на количество переменных с использующих такой тип:

```
class PairOfT<T> {
    T object1;
    T object2;

    PairOfT(T one, T two) {
        object1 = one;
        object2 = two;
    }

    public T getFirst() {
        return object1;
    }

    public T getSecond() {
        return object2;
    }
}
```

Алмазный синтаксис (Diamond syntax)

Вернёмся немного назад к примеру со строкой кода:

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

Если типы не будут совпадать:

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

То мы получим ошибку при компиляции:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
    The constructor Pair<String,String>(int, String) is undefined
    Type mismatch: cannot convert from Pair<String,String> to
Pair<Integer,String>

    at test.Test.main(Test.java:23)
```

Немного лениво каждый раз заполнять типы и при этом можно ошибиться. 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"); // Error
```

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 вот этот 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());
```

Теперь посмотрим на вот этот:

```
    List<String> list = new LinkedList<String>();

    list.add("First");
    list.add("Second");

    List 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. В общем, это очень похоже на 2 самых первых примера. Если в двух словах, то 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`.)

Рассмотрим реализацию такого метода:

```
package test;

import java.util.ArrayList;
import java.util.List;

class Utilities {
    public static <T> void fill(List<T> list, T val) {
        for (int i = 0; i < list.size(); i++)
            list.set(i, val);
    }
}

class Test {
    public static void main(String[] args) {
        List<Integer> intList = new ArrayList<Integer>();

        intList.add(1);
        intList.add(2);

        System.out.println("Список до обработки дженерик-методом: " +
intList);

        Utilities.fill(intList, 0);

        System.out.println("Список после обработки дженерик-методом: " +
intList);
    }
}
```

Нам в первую очередь интересно это:

```
public static <T> void fill(List<T> list, T val)
```

`<T>` размещено после ключевых слов `public` и `static`, а затем следуют тип возвращаемого значения, имя метода и его параметры. Такое объявление отлично от объявления универсальных классов, где универсальный параметр указывается после имени класса. Тело метода вполне обычное – в цикле все элементы списка устанавливаются в одно значение (`val`). Ну и в `main()`-методе происходит вызов нашего универсального метода:

```
Utilities.fill(intList, 0);
```

Стоит обратить внимание на то, что здесь не задан явно тип параметра. Для `IntList` – это `Integer` и 100 тоже упаковывается в `Integer`. Компилятор ставит в соответствие типу `T` – `Integer`.

Возможны ошибки, связанные с импортом `List` из `java.awt` вместо `java.util`. Важно помнить, что список из `java.util` является универсальным типом а список из `java.awt` - нет.

А сейчас вопрос – какая(-ие) из нижеприведённых строк скомпилируется без проблем?

```
1. List<Integer> list = new List<Integer>();
2. List<Integer> list = new ArrayList<Integer>();
3. List<Number> list = new ArrayList<Integer>();
4. List<Integer> list = new ArrayList<Number>();
```

Перед ответом на этот вопрос следует учесть, что `List` – интерфейс, `ArrayList` наследуется от `List`; `Number` - абстрактный класс и `Integer` наследуется от `Number`.

~Ответ с пояснением:~

- Первый вариант неправильный, т.к. нельзя создавать объект интерфейса.
- Во втором случае мы создаем объект типа ArrayList и ссылку на него базового для ArrayList класса. И там, и там дженерик-тип одинаковый – всё правильно.
- В третьем и четвёртом случае будет иметь ошибка компиляции, т.к. дженерик-типы должны быть одинаковыми (связи наследования здесь никак не учитываются).

Условие одинаковости дженерик-типов может показаться не совсем логичным. В частности хотелось бы использовать конструкцию под номером 3. Почему же это не допускается?

Будем думать от обратного – допустим 3-ий вариант возможен. Рассмотрим такой код:

```
/*
* Данный код не скомпилируется из-за первой строки. На его примере
* объясняется, почему он не должен компилироваться
*/
List<Number> intList = new ArrayList<Integer>();

intList.add(new Integer(10));
intList.add(new Float(10.0f)); // (3)
```

Первая строка кода смотрится вполне логично, т.к. `ArrayList` наследуется от `List` , а `Integer` наследуется от `Number`. Однако допуская такую возможность мы получили бы ошибку в 3 строке этого кода, ведь динамический тип `IntList` - `ArrayList <Integer>`, т.е. происходит нарушение типобезапасности (присвоение значение `Float` там, где ожидается `Integer`) и в итоге была бы получена ошибка компилятора. Дженерики созданы, чтобы избегать ошибок такого рода, поэтому существует данное ограничение. Но тем не менее это неудобное ограничение и Java поддерживает маски для его обхода.

Wildcards (Маски)

Сейчас будут рассмотрены Wildcard Parameters (wildcards). Этот термин в разных источниках переводится по-разному: метасимвольные аргументы, подстановочные символы, групповые символы, шаблоны, маски и т.д. В данной статье я буду использовать "маску", просто потому, что в ней меньше букв...

Как было написано выше вот такая строка кода не скомпилируется:

```
    List<Number> intList = new ArrayList<Integer>();
```

Но есть возможность похожей реализации:

```
List<?> intList = new ArrayList<Integer>();
```

Под маской мы будем понимать вот эту штуку – `<?>`.

А сейчас пример кода использующего маску и пригодного к компиляции:

```
class Test {
    static void printList(List<?> list) {
        for (Object l : list)
            System.out.println("{" + l + "}");
    }

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(100);
        printList(list);

        List<String> strList = new ArrayList<>();
        strList.add("10");
        strList.add("100");
        printList(strList);
    }
}
```

Метод `printList` принимает список, для которого в сигнатуре использована маска:

```
    static void printList(List<?> list)
```

И этот метод работает для списков с различными типами данных (в примере `Integer` и `String`).

Однако вот это не скомпилируется:

```
List<?> intList = new ArrayList<Integer>();

intList.add(new Integer(10));
/* intList.add(new Float(10.0f)); даже с закомментированной последней
строкой не скомпилируется */
```

Почему не компилируется? При использовании маски мы сообщаем компилятору, чтобы он игнорировал информацию о типе, т.е. <?> - неизвестный тип. При каждой попытке передачи аргументов дженерик-типа компилятор Java пытается определить тип переданного аргумента. Однако теперь мы используем метод `add ()` для вставки элемента в список. При использовании маски мы не знаем, какого типа аргумент может быть передан. Тут вновь видна возможность ошибки, т.к. если бы добавление было возможно, то мы могли бы попытаться вставить в наш список, предназначенный для чисел, строковое значение. Во избежание этой проблемы, компилятор не позволяет вызывать методы, которые могут добавить невалидный тип - например, добавить значение типа `Float`, с которым мы потом попробуем работать как с `Integer` (или `String` — по маске не определишь точно). Тем не менее есть возможность получить доступ к информации, хранящейся в объекте, с использованием маски, как это было показано выше.

И ещё один маленький пример:

```
List<?> numList = new ArrayList<Integer>();
numList = new ArrayList<String>();
```

Тут не возникнет проблем компиляции. Однако нехорошо, что переменная `numList` хранит список со строками. Допустим нам нужно так объявить эту переменную, чтобы она хранила только списки чисел. Решение есть:

```
List<? extends Number> numList = new ArrayList<Integer>();
numList = new ArrayList<String>();
```

Данный код не скомпилируется, а всё из-за того, что с помощью маски мы задали ограничение. Переменная `numList` может хранить ссылку только на список, содержащий элементы унаследованные от `Number`, а всё из-за объявления: `List<? extends Number> numList`. Тут мы видим, как маске задаётся ограничение – теперь `numList` предназначен для списка с ограниченным количеством типов. `Double` как и `Integer` наследуется от `Number`, поэтому код приведённый ниже скомпилируется.

```
List<? extends Number> numList = new ArrayList<Integer>();
numList = new ArrayList<Double>();
```

То, что было описано выше называется ограниченными масками (~Bounded wildcards~). Применение таких конструкций может быть весьма красивым и полезным. Допустим нам необходимо посчитать сумму чисел различного типа, которые хранятся в одном списке:

```
public static Double sum(List<? extends Number> numList) {
    Double result = 0.0;
    for (Number num : numList) {
        result += num.doubleValue();
    }
    return result;
}
```

`Double`-тип был использован для переменной `result` т.к. он без проблем взаимодействует с другими числовыми типами (т.е. не будет проблем с приведением типов).

В завершение этой темы добавлю, что аналогично ключевому слову `extends` в подобного рода выражениях может использоваться ключевое слово `super` — `<? super Integer>`. Выражение `<? super X>` означает, что вы можете использовать любой базовый тип (класс или интерфейс) типа `X`, а также и сам тип `X`. Пара строк, которые нормально скомпилируются:

```
List<? super Integer> intList = new ArrayList<Integer>();

System.out.println("The intList is: " + intList);
```