Виталий Унгурян
unguryan@itstep.org

Обобщённое программирование — это такой подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания.
Generics (дженерики) или <<контейнеры типа T>> — подмножество обобщённого программирования.

Обобщения - это параметризованные типы. С их помощью можно объявлять классы, интерфейсы и методы, где тип данных указан в виде параметра.

Обобщения - добавили в язык java безопасность типов.

```
class Box {
  private Object value;
  public Box(Object value) {
    this.value = value;
  }
  public Object get() {
     return value;
  }
}
```

```
class Box <E>{
  private E value;
  public Box(E value) {
    this.value = value;
  }
  public E get() {
     return value;
  }
}
```

После имени класса в угловых скобках "<" и ">" указано имя типа “E", которое может использоваться внутри класса.

Фактически E – это тип, который должен быть определён позже (при создании объекта класса).

В именах переменных типа принято использовать заглавные буквы.

Обычно для коллекций используется буква **E**, буквами **K** и **V** - типы ключей и значение (Key/Value), а буквой **T** (и при необходимости буквы **S** и **U**) - любой тип.

```
Box<Tea>  box1 =
new  Box<Tea>(new Tea());
Tea tea = box1.get();

Box<Coffee>  box 2 =
new  Box<Coffee> (new Coffee ());
Coffee tea = box2.get();
```

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");

Для того чтобы сохранить целостности и независимости друг от друга, у Generics существует так называемая "Несовместимость generic-типов".

```
List<Integer> li = new ArrayList<Integer>();
List<Object> lo = li;
lo.add(“hello”);
```

Невозможно создать объект generic типа, поскольку компилятор не знает, какой конструктор вызвать.

```
private static <T> T get(T value) {
   return new T();
}
```

Невозможно реализовывать одновременно два одинаковых интерфейса с разными типами.

```
public class DecimalString implements
Comparable<Number>,
Comparable<String> {}
```

Невозможно объявить статическое поле generic типа

```
public class MyClass<T> {
   private static T value;
}
```

Невозможно использовать instatceof для параметризованного типа.

```
public static <E> void setList(List<E> list) {
   if (list instanceof ArrayList<Integer>) {}
}
```

Невозможно создать массив параметризированного типа

```
Box<Tea>[] arrayOfLists = new
Box<Tea>[2];
```

Невозможно перегрузить метод, в котором типы параметров “стираются” до одного и того же типа.

```
public void print(Set<String> strSet) { }
public void print(Set<Integer> intSet) { }
```

Wildcard Parameters (wildcards).

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

Шаблон аргументов указывается символом **?** и представляет собой неизвестный тип.

```
Object obj = new Object();
 Box<?> box3 = new Box<Object>(obj );
```

Одно из преимуществ wildcards состоит в том, что они дают возможность написать код, который может оперировать различными generic-типами без знания их точных границ.

```
public void unbox(Box<?> box) {
    System.out.println(box.get());
}
```

```
public void rebox(Box<?> box) {
  box.put(box.get());
}
```

```
public void rebox(Box<?> box) {
   reboxHelper(box);
}
private<V> void reboxHelper(Box<V> box) {
  box.put(box.get());
}
```

Вспомогательный метод reboxHelper() является generic-методом. Generic-методы вводят дополнительные параметры типов (помещаемые в угловые скобки перед типом возвращаемого значения), которые обычно используются для формулирования ограничений типов между параметрами и/или возвращаемым значением метода.

Однако в случае reboxHelper() generic-метод не задействует параметр типа для определения ограничения типа, а позволяет компилятору – через вывод типа – дать имя параметру типа переменной box. Приём с capture-хелпером основан на выводе типов (type inference) и преобразовании при фиксации (capture conversion).

```
Box<? extends Coffee> box3 = new
Box<Coffee>(new Coffee());

Box<? extends Tea> box3 = new
Box<Tea>(new Tea());
```

```
Box<? super Coffee> box3 = new
Box<Coffee>(new Coffee());
```

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 generic-классе. Универсальные методы удобны, когда одна и та же функциональность должна применяться к различным типам.

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

Поддержка generic-ов реализована средствами компилятора. Виртуальная машина не предоставляет никакой поддержки generic-ов, кроме возможности получения информации о типах.

Во время компиляции generic-параметры убираются, и, там, где это требуется, вместо них вставляется приведение типов.