БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра веб-технологий и компьютерного моделирования

А. С. Кравчук, А. И. Кравчук, Е. В. Кремень

ЯЗЫК JAVA
Дженерики

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

МИНСК
2023

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1
К78

Рекомендовано советом
механико-математического факультета БГУ
26 января 2023 г., протокол № 5

Рецензент
кандидат технических наук, доцент *М. Н. Садовская*

Кравчук, А. С.
К78 Язык Java. Дженерики : учеб. материалы для студентов спец. 1-31 03 08 «Математика и информационные технологии (по направлениям)» / А. С. Кравчук, А. И. Кравчук, Е. В. Кремень. – Минск : БГУ, 2023. – 50 с.

Рассматриваются вопросы использования дженериков, или обобщений, для включения типов в качестве параметров при определении классов, интерфейсов и методов. Несмотря на кажущуюся простоту дженериков, многие сталкиваются с трудностями при их использовании, поэтому подробно обсуждаются преимущества, ограничения и побочные эффекты использования обобщений. Приводится необходимый теоретический материал и код программ, что существенно ускоряет усваивание материала, а также способствует более квалифицированному подходу к программированию. Издание ориентировано как на тех, кто не имеет опыта практического программирования на языке Java, так и на тех, кто хотел бы систематизировать и улучшить свои знания.

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1

© Кравчук А. С., Кравчук А. И.,
Кремень Е. В., 2023
© БГУ, 2023

Оглавление

Параметризованные методы классов 5
Методы с частично параметризованными типами 9
Параметризация конструктора 10
Перегрузка параметризованных методов 11
Удобство применения параметризованных методов 12
Недостатки параметризованных методов 12
Параметризованные классы 13
Бриллиантовая операция (Diamond operator) 14
Соглашение об именовании переменных типа 14
Параметризованный класс с экземплярными методами 15
Статические методы в параметризованном классе 16
Метасимвольный аргумент (wildcards или подстановочный символ/знак) 18
Класс с несколькими параметрами 20
Использование в качестве параметра класса параметризованного типа 21
Сырой тип (Raw type) 22
Наследование в обобщенных типах 24
Обобщенные типы в качестве ограничений 29
Преимущества использования параметризованных классов 29
Пример класса демонстрирующий, что параметризация не всегда нужна 30
Параметризация интерфейса 31
Реализация параметризованного интерфейса 32
Реализация параметризованным классом интерфейса с большим числом параметров чем у класса 34
Доступ к реализациям объектов через ссылки на параметризованный интерфейс 35
Вложенные (внутренние) параметризованные интерфейсы 38
Расширение параметризованных интерфейсов 40
Реализация множества параметризованных интерфейсов в классах 42

Дефолтные параметризованные методы 43
Параметризованные статические методы 44
Приватные параметризованные методы...46
К вопросу о приведении типов при реализации обобщений........................48
Дополнительные сведения по обобщениям 49
Выведение типов...49
Целевые типы 49
Стирание типа (`Type Erasure`)...50
Литература 50

Параметризованные методы классов

Достаточно часто встречаются задачи, в которых одна и та же операция выполняется с различными типами данных. При этом в языке Java до изучения настоящего раздела приходилось использовать несколько разных методов с разными именами, каждый из которых работает со своим типом данных. Таким образом, до изучения текущего раздела разработчик мог решить, хотя и неэффективно, данную проблему с помощью перегрузки методов.

Если текст методов за исключением типов совпадает, то язык Java предлагает средство, позволяющее упростить работу разработчика — создать один метод, работающий с различными типами данных.

Такой метод называется параметризованным методом (иногда шаблоном, но чаще `generic`-методом). Создание шаблона метода аналогично созданию обычного метода, но заголовок параметризованного несколько сложнее.

Первый вариант описания параметризованного метода в случае *ограничения сверху* на параметризуемый тип имеет вид:

```
спец <Тип extends Класс> типЗнач имяМетода(Тип имяПарам1,
                                           …,
                                           Тип имяПарамN){
    //тело метода
}
```

где ~~спец~~ – спецификатор `static` и/или модификатор доступа.

В данном случае **Тип** может принимать в качестве значения классы, расширяющие (`extends`) суперкласс **Класс**, а также сам тип **Класс**. «Ограничение сверху» означает, что класс **Класс** находится в вершине иерархии классов наследников, обозначаемых через **Тип**.

В случае *ограничения снизу* на параметризуемый тип описание параметризованного метода приобретает вид:

```
спец <Тип super Класс> типЗнач имяМетода(Тип имяПарам1,
                                         …,
                                         Тип имяПарамN){
    //тело метода
}
```

В данном случае **Тип** может принимать в качестве значения классы, являющиеся предками, т.е. супер-типами для типа **Класс**, а также сам тип

Класс. «Ограничение снизу» означает, что класс **Класс** находится в корне иерархии классов наследников, обозначаемых через **Тип**.

Параметр `типЗнач` – определяет тип возвращаемого значения, а `имяМетода` – свободно выбираемый идентификатор.

Список **Тип** `имяПарам1, …,` **Тип** `имяПарамN` может иметь произвольное количество формальных параметров, но все они должны быть одного параметризованного типа **Тип**.

После заголовка должно идти обычное описание метода, с использованием параметрического названия (псевдонима **Тип**) параметризованного типа.

При использовании параметризованного метода в программе, компилятор подставит вместо указанного имени типа реально используемый тип.

Пример.

```
1   public class Program
2   {
3       //T может принимать значения типов любых числовых
4       //классов-оберток, расширяющих Number
5       public static <T extends Number> T max(T a, T b) {
6         if( a.doubleValue() > b.doubleValue()) return a;
7         return b;
8       }
9
10      public static void main(String [] args) {
11        System.out.println(max(1, 5));
12        System.out.println(max(2, 2.5));
13        //ошибка компиляции
14        //System.out.println(max('a', 'b'));
15      }
16  }
```

Результат работы программы:

```
5
2.5
```

Очевидно, что компилятор просто подставляет нужный тип в вместо параметра `T` в параметризованном методе. Однако он будет не всегда работоспособен. Хотя создается впечатление, что данный алгоритм может быть определен независимо от типа данных, но он обязательно пользуется свойствами этих данных. В случае с параметризованным методом **max** это

требование определения операции сравнения (операция «>»). Именно поэтому внутри параметризованного метода используется экземплярный метод doubleValue(), приводящий любое из значений классов оберток к базовому типу double, для которого определена операция сравнения «больше».

Таким образом, любой параметризованный метод предполагает наличие определенных свойств у значений параметризуемого типа, в зависимости от реализации (например, наличия определенных для данного типа (или класса), метода (или конструктора), операции сравнения и т.д.).

<u>*Второй вариант*</u> описания параметризованного метода – это вариант без ограничений на количество параметризованных типов:

```
спец <Тип1,…,ТипN> типВозврЗнач имяМетода(списПарам) {
    //тело метода
}
```

где ~~спец~~ – спецификатор static и/или спецификатор доступа; типы Тип$_1$,…, Тип$_N$ – типы данных, передаваемых в параметризованный метод, которые будут неявно определяться при вызове параметризованного метода в соответствии с передаваемыми фактическими значениями; списокПарам - это перечисление через запятую выражений вида $\{\textbf{Тип}_i\ \text{имяПарам}_j\}_{(i=\overline{1,N},j=\overline{1,M})}$, (где $N \leq M$).

Очевидно, что если параметризуется только один тип, то и список параметров сокращается до одного.

Как и в первом случае после заголовка должно идти обычное описание метода, с использованием параметрических названий (псевдонимов Тип$_1$,…, Тип$_N$) перечисленных типов.

<u>*Замечание.*</u>

Ключевые слова extends/super могут применяться в произвольном числе параметров от общего их числа, использованных при параметризации метода.

<u>*Пример.*</u>

```
1 public class GenMethodDemo {
2     public static <T, V> boolean isIn(T x, V[] array) {
3         for (V element : array) {
4             //далее в if() необходимо использовать
5             //переопределенный метод equals() класса
6             //Object для сравнения объектов.
7             if (((Object) x).equals((Object) element)) {
```

```
 8                 return true;
 9             }
10             //Приведение типов в if() ((Object) x) и
11             //((Object) element) излишни, т.к. выполняются
12             //автоматически, т.к. заголовок метода equals()
13             //имеет вид Object equals(Object x). Именно
14             //автоматическое "восходящее" преобразование
15             //типов дает возможность сравнивать объекты
16             //разных классов
17         }
18         return false;
19     }
20 
21     public static void main(String[] args) {
22         Integer[] intArray = {1, 2, 3, 4, 5};
23 
24         if (isIn(2, intArray)) {
25           System.out.println("2 входит в массив intArray");
26         }
27         if (!isIn(7, intArray)) {
28           System.out.println("7 не входит в intArray");
29         }
30         System.out.println();
31 
32         //объявляем массив значений типа String
33         String[] strArray = {"one", "two", "three"};
34         if (!isIn('a', strArray)) {
35           System.out.println("a не входит в массив строк");
36         }
37     }
38 }
```

Результат работы программы:

```
2 входит в массив intArray
7 не входит в intArray

a не входит в массив строк
```

Замечание.

При вызове параметризованного метода в качестве фактического значения аргумента передается число 2 (базовый тип `int`*), но «по умолчанию» фактически на вход параметризованного метода приходит объект класса-обертки* `Integer` *со значением 2. Соответственно при*

передаче символа `'a'` *также на вход метода приходит объект класса-обертки. Именно поэтому во всех случаях можно вызвать экземплярный метод* `equals()`.

При работе с обобщенными типами нельзя использовать элементарные базовые типы. Можно использовать только классы-«обертки» над примитивными базовыми типами. При этом, при указании значений не нужно явно создавать экземпляр класса-обертки. Можно просто указать значение, и компилятор автоматически «упакует» его в экземпляр соответствующего класса:

Как видно из примеров параметризация типов параметров метода класса никак не связана с параметризацией самого класса. Класс при этом *может не* быть параметризованным как в примерах выше.

В общем случае методы с параметризованными типами следует использовать «по мере возможности». Иначе говоря, если возможно параметризовать метод вместо целого класса, вероятно, стоит выбрать именно этот вариант.

Замечание.

В отличие от экземплярных статические методы не имеют доступа к параметрам типа классов-дженериков. Если такие методы должны использовать параметризацию (см. примеры выше), то это должно происходить на уровне метода, а не на уровне класса.

Методы с частично параметризованными типами

В методах с частично параметризованными типами можно в заголовке смешивать параметры с ***не***параметризованными и параметризованными типами. Эти параметры работают так же, как в любом другом методе.

Пример.

```
1 public class GenMethodDemo {
2     public static <T> boolean isIn(double x, T[] array) {
3         for (T element : array) {
4             if (element.equals(x)) {
5                 return true;
6             }
7         }
```

```
8           return false;
9       }
10
11      public static void main(String[] args) {
12          String[] strArray = {"one", "two", "three"};
13          if (!isIn(1.7, strArray)) {
14              System.out.println("1.7 не входит " +
15                                      "в массив строк");
16          }
17      }
18  }
```

Результат работы программы:

```
1.7 не входит в массив строк
```

Замечания:

- *аргументов с непараметризованными типами, как и с параметризованными, в частично параметризованном методе может быть любое количество;*
- *не смотря на автоматическое преобразование значений аргументов базовых типов в объекты классов-оберток при вызове методов рекомендуется явно указывать именно классы-обертки в качестве типов непараметризованных аргументов.*

Параметризация конструктора

В ***не***параметризованном классе можно параметризовать только конструктор.

Пример.

```
1  class GenConstructor {
2      private double value;
3      //параметризованный конструктор в
4      //непараметризованном классе
5      public <T extends Number> GenConstructor(T arg) {
6          value = arg.doubleValue();
7      }
8
9      public double getValue() {
10         return value;
```

```
11        }
12    }
13
14    public class Main {
15        public static void main(String[] args) {
16            GenConstructor generation1 =
17                                new GenConstructor(100);
18            GenConstructor generation2 =
19                                new GenConstructor(123.5F);
20
21            System.out.println("value: " +
22                                generation1.getValue());
23            System.out.println("value: " +
24                                 generation2.getValue());
25        }
26    }
```

Результат работы программы:

```
value: 100.0
value: 123.5
```

Если необходимо параметризовать конструктор без параметров, то

Перегрузка параметризованных методов

Помимо создания явным образом перегруженных версий обычных методов, можно также перегружать спецификацию параметризованных методов. Для этого достаточно создать еще одну версию параметризации, которая будет отличаться от остальных списком параметров.

Пример.

```
 1  public class GenMethodDemo {
 2      //первая параметризация метода metod()
 3      public static <T extends Number > String method(T x) {
 4         return x.toString();
 5      }
 6      //вторая параметризация метода metod()
 7      public static <T extends Number, V extends Number>
 8                                String method(T x, V y) {
 9         return x.toString() + " "+ y.toString();
10      }
```

```
11
12      public static void main(String[] args) {
13          System.out.println(method(2));
14          System.out.println(method(2, 3.3));
15      }
16  }
```

В примере выше параметризованный для метод `method()` перегружается, чтобы обеспечить возможность приема как одного, так и двух параметров. Результат работы программы:

```
2
2 3.3
```

Удобство применения параметризованных методов

Параметризованные методы экономят много времени, так как параметризованный метод создается только один раз, а использовать его можно с разными типами данных. По времени создание параметризованного метода занимает не больше времени, чем написание обычного. Параметризованный метод намного упрощают дальнейшую поддержку кода.

Недостатки параметризованных методов

У параметризованных методов есть несколько недостатков:

- параметризованные методы могут выдавать сложные и длинные сообщения об ошибках, которые намного сложнее расшифровать, чем ошибки, вызываемые обычными методами;
- параметризованные методы могут увеличить время компиляции и размер кода, так как один параметризованный может быть «реализован» и перекомпилирован в нескольких файлах;
- хотя допускается использование параметризации и для методов имеющих список параметров переменной длины, но их использование может привести к «загрязнению» кучи.

Однако перечисленные недостатки незначительны по сравнению с мощью и гибкостью параметризованных методов.

Параметризованные классы

Язык Java позволяет использовать параметризацию не только для методов, но и для классов. Если есть необходимость в классах, работающих одинаково, но содержащих значения разных типов, можно воспользоваться возможностями параметризации.

Формат объявления параметризованного класса ИмяGeneric:

```
~~специф~~ class ИмяДженерика <ИмяТипа1,…, ИмяТипаN> {
    ... // тело класса
}
```

где ~~специф~~ – спецификатор доступа; ИмяТипа1, …, ИмяТипаN – обобщенные имена типов (свободно выбираемые идентификаторы), которые используются методами и свойствами класса; **ИмяДженерика** – свободно выбираемый идентификатор (имя параметризованного класса).

Замечания:

- *в списке перечисления параметров класса допустимо использовать ключевые слова extends/super в уже известной синтаксической конструкции: ИмяТипа extends/super Тип;*
- *как и ранее extends указывает на то, что получаемый фактический тип ИмяТипа расширяет явно заданный суперкласс Тип, а слово super указывает на то, что в качестве ИмяТипа будут использовать исключительно предки класса Тип;*
- *ключевые слова extends/super могут применяться в произвольном числе параметров от общего их числа, использованных при параметризации класса.*

Формат объявления объекта параметризованного (обобщенного) класса с одним параметром имеет следующий вид:

```
ИмяДженерика<ИмяТипа> ОбъектДжен;
```

где **ИмяДженерика** – имя параметризованныйного класса; **ИмяТипа** – конкретный тип данных в программе; ОбъектДжен – имя объекта (экземпляра) класса.

Инициализация объекта параметризованного класса с одним параметром имеет один из следующих форматов:

- одна инструкция:

```
ИмяGeneric <ИмяТипа> ОбъектДжен =
        new ИмяGeneric <>(new ИмяТипа(парамКонстрТипа));
```

- две инструкции с предварительной инициализацией объекта, используемого во второй ***уже*** для инициализации generic-а:

```
ИмяТипа ОбъектИниц =new ИмяТипа(парамКонстрТипа);
ИмяGeneric <ИмяТипа> ОбъектДжен =
                    new ИмяGeneric <>(ОбъектИниц);
```

Замечания:

- *дженерики работают только с объектами. Поэтому в качестве параметров должны использоваться только классы или если необходимо использовать базовые типы, то их классы-обертки;*
- *дженерик-поля классов не могут быть статическими;*
- *если в заголовке метода параметризованного класса перечисляются некоторые параметры (пусть даже совпадающие по именам с параметрами класса), то компилятор будет считать эти два набора параметров разными (не связанными друг с другом).*

Бриллиантовая операция (Diamond operator)

Начиная с Java 7 существует также бриллиантовая операция (`diamond operator`), которая позволяет указывать пустые аргументы типа `<>` там, где компилятор может вывести тип из контекста. В частности, при инициализации объекта параметризованного класса справа:

```
ИмяGeneric <ИмяТипа> ОбъектGen =
       new ИмяGeneric<> (new ИмяТипа(парамКонстрТипа));
```

Соглашение об именовании переменных типа

По соглашению переменные типа именуются одной буквой в верхнем регистре. Это сильно отличается от соглашения об именовании переменных, классов и интерфейсов. Без такого отличия было бы трудно отличить переменную типа от класса и интерфейса.

Наиболее часто используемые имена для параметров типа:

- `E` — элемент (`Element`, обширно используется `Java Collections Framework`);
- `K` — ключ;
- `N` — число;
- `T` — тип;
- `V` — значение;
- `S`, `U`, `V` и т. п. — 2-й, 3-й, 4-й типы.

Параметризованный класс с экземплярными методами

Особенно важным является то, что в качестве значений параметров классов могут использоваться любые объекты, в частности, записи (`records`), которые должны быть подготовлены соответствующим образом (т.е. должны быть переопределены все необходимые методы).

Пример.

```
1   record BaseInform (String name,
2                      char gender,
3                      int age) {}
4
5   record Child (BaseInform inf, int iq) {
6       @Override
7       public String toString() {
8           return "Child: " + inf.name() +
9                  " IQ: " + iq ;
10      }
11  }
12
13  class GenericClass <T> {
14      private T[] array;
15      //конструктор
16      public GenericClass(T[] a) {
17          this.array = a.clone();
18      }
19      //экземплярные методы
20      public T[] getData() {
21          return array;
22      }
```

```
23      public void setData(T s, int i) {
24          array[i] = s;
25      }
26      @Override
27      public String toString() {
28          String str =  array[0].toString() + "\n";
29          for(int i = 1; i <array.length; i++) {
30              str = str + array[i].toString() + "\n";
31          }
32          return str;
33      }
34  }
35
36  public class Main
37  {
38      public static void main(String[] args) {
39          Child[] a =
40          {new Child(
41              new BaseInform("Sidorov", 'm', 15), 80),
42          new Child(
43              new BaseInform("Ivanov", 'm', 5), 120)};
44          GenericClass<Child> array =
45                                  new GenericClass<>(a);
46          System.out.println(array.toString());
47      }
48  }
```

Результат выполнения программы:

```
Child: Sidorov IQ: 80
Child: Ivanov IQ: 120
```

Статические методы в параметризованном классе

Приведем пример того, как должен выглядеть статический метод в рамках класса `Array` с параметром `T`:

Пример.

```
1  import java.util.Arrays;
2
3  class GenericClass <T extends Number> {
```

```
4       private T[] array;
5       //конструктор
6       public GenericClass(T[] a) {
7           this.array = Arrays.copyOf(a, a.length);
8       }
9       //экземплярные методы
10      public T[] getData() {
11          return array;
12      }
13      public Double average() {
14          Double sum = 0.0;
15          for (T value : array) {
16              sum += value.doubleValue();
17          }
18          return sum / array.length;
19      }
20      //перегрузка метода toString() класса Object
21      public static String toString(GenericClass a) {
22        return Arrays.toString(a.getData());
23      }
24  }
25
26  public class Main
27  {
28     public static void main(String[] args) {
29        Double[] a = {1., 2., 3., 4., 5.};
30        GenericClass <Double> array =
31                            new GenericClass<>(a);
32        System.out.println(GenericClass.toString(array));
33        System.out.println("Среднее число " +
34                                       array.average());
35     }
36  }
```

Результат работы программы:

```
[1.0, 2.0, 3.0, 4.0, 5.0]
Среднее число 3.0
```

Метасимвольный аргумент (wildcards или подстановочный символ/знак)

Допустим, что необходимо добавить метод для сравнения *средних значений* массивов в класс `GenericClass` из предыдущего примера. Причем типы массивов должны быть разные. Таким образом хотелось бы организовать с помощью экземплярного метода `isTheSame()` выполнение следующего алгоритма в `main()`:

```
Integer intArray[] = {1, 2, 3, 4, 5};
Double doubleArray[] = {1.1, 2.2, 3.3, 4.4, 5.5};
GenericClass <Integer> i =
                    new GenericClass <>(intArray);
GenericClass <Double> d =
                    new GenericClass <>(doubleArray);
if (i.isTheSame(d)) System.out.println("Совпадают");
else System.out.println("Различаются.");
```

Замечание.

Напомним, что для вычисления средних значений массивов в предыдущем примере используется метод `average()`, который вне зависимости от типа числового массива возвращает среднее значение единственно возможного и в данном случае универсаньного типа `Double`. Именно поэтому сравнение средних значений массивов разных числовых типов возможно.

Так как `GenericClass` параметризованный тип, то встает неизбежный вопрос какой тип параметра необходимо указать для `GenericClass`, в заголовке экземплярного метода `isTheSame(d)`. Напрашивается следующий вариант:

```
boolean isTheSame(GenericClass <T> obj) {
    return this.average().equals(obj.average());
}
```

Но это не сработает, так как в этом случае метод `isTheSame` будет принимать аргументы только того же типа, что и существующий объект `this`, который и вызывает этот метод (в примере выше это объект `i`, т.к. `i.isTheSame(d)`).

Чтобы создать обобщенную версию метода `isTheSame`, следует воспользоваться другим средством обобщений Java – метасимвольным

аргументом. Метасимвольный аргумент обозначается знаком **?** и представляет неизвестный тип.

Пример.

```
boolean isTheSame(GenericClass <?> obj) {
    return this.average().equals(obj.average());
}
```

Метасимвол не оказывает никакого влияния на тип создаваемых объектов класса `GenericClass`. Метасимвол просто совпадает с любым достоверным объектом класса `GenericClass`.

Пример.

```
1   import java.util.Arrays;
2
3   class GenericClass <T extends Number> {
4       private T[] array;
5       //конструктор
6       public GenericClass(T[] a) {
7           this.array = Arrays.copyOf(a, a.length);
8       }
9       //экземплярные методы
10      public T[] getData() {
11          return array;
12      }
13      public Double average() {
14          Double sum = 0.0;
15          for (T value : array) {
16              sum += value.doubleValue();
17          }
18          return sum / array.length;
19      }
20      boolean isTheSame(GenericClass <?> obj) {
21          return this.average().equals(obj.average());
22      }
23      //статический метод
24      public static String toString(GenericClass a) {
25          return Arrays.toString(a.getData());
26      }
27  }
28
29  public class Main
30  {
31      public static void main(String[] args) {
32          Integer intArray[] = {1, 2, 3, 4, 5};
```

```
33            Double doubleArray[] = {1., 2., 3., 4., 5.};
34            GenericClass <Integer> i =
35                    new GenericClass <>(intArray);
36            GenericClass <Double> d =
37                    new GenericClass <>(doubleArray);
38            if (i.isTheSame(d)) {
39                System.out.println("Средние совпадают");
40            }
41            else {
42                System.out.println("Средние различаются");
43            }
44        }
45    }
```

Результат работы программы:

```
Средние совпадают
```

Замечание.
Метасимволы применимы и к generic-методам.

Метасимвольные аргументы могут быть ограничены таким же образом, как и параметры типов. Ограничивать метасимвольный аргумент особенно важно при создании обобщенного типа, оперирующего иерархией классов. Допускаются следующие виды ограничений:

- `<? extends T>` - называется «подстановочный знак с ограничением сверху», пример: `<? Extends Number>`;
- `<?>` - реализация «по умолчанию»: `<? extends Object>`. Запись означает: «любой тип унаследованный от `Object`», т.е. просто «любой тип»;
- `<? super T>` - называется «подстановочный знак с ограничением снизу», пример `List<? super Integer>`.

Замечание.
Не следует использовать подстановочные символы в возвращаемых типах, потому что это будет принуждать других программистов, работающих в проекте долго разбираться с подобным синтаксисом.

Класс с несколькими параметрами

Обобщенный тип (класс) может иметь несколько параметров типа.

Пример.

```java
class Goblin {}
class Genie {}

class PairLair<T, S> {
    T obj1;
    S obj2;
    //методы
    public void setObj1(T obj1) {
        this.obj1 = obj1;
    }
    public T getObj1() {
        return this.obj1;
    }
    public void setObj2(S inhabitant2) {
        this.obj2 = inhabitant2;
    }
    public S getObj2() {
        return this.obj2;
    }
}

public class Main
{
    public static void main(String[] args) {
        PairLair<Goblin, Genie> goblinGenie =
                          new PairLair<>();
        goblinGenie.setObj1(new Goblin());
        goblinGenie.setObj2(new Genie());
        Goblin goblin = goblinGenie.getObj1();
        Genie genie = goblinGenie.getObj2();
    }
}
```

Данная программа не выводит никаких результатов, а написана только для демонстрации возможного синтаксиса и будет успешно скомпилирована и выполнена.

Использование в качестве параметра класса параметризованного типа

Можете также заменить параметр типа на параметризованный тип:

```
PairLair<Goblin, Lair<Genie>> hierarchicalLair =
                                    new PairLair<>();
```

Обобщенные интерфейсы объявляются схожим с обобщенными классами способом.

Сырой тип (Raw type)

Сырой тип (raw type) — это имя обобщенного класса или интерфейса без аргументов типа (type arguments). Формат создания объекта сырого типа:

```
ИмяGeneric ОбъектДжен; //ИмяGeneric без параметров
```

Инициализация объекта сырого типа:

```
ИмяGeneric ОбъектДжен = new ИмяGeneric <>();
```

Например, если в синтаксической конструкции `PairLair<Goblin, Genie> obj = new PairLair<>();` убрать аргументы типа (`<Goblin, Genie>`), то будет создан объект `obj` сырого типа с помощью следующей инструкции: `PairLair obj = new PairLair();`. Таким образом `PairLair` — это сырой тип обобщенного типа `PairLair<T, S>`.

Замечание.

*Обычный необобщенный класс или интерфейс **НЕ** являются сырыми типами.*

Допустимо присваивать объект параметризованного типа `obj` объекту `obgRaw` своего собственного сырого типа.

Пример.

```
//описание параметризованного класса PairLair и двух
//классов-маркеров Goblin и Genie такие же, как и
//раньше
public class Main
{
    public static void main(String[] args) {
        PairLair<Goblin, Genie> obj =
                              new PairLair<>();
```

```
        PairLair obgRaw = obj;
    }
}
```

Можно сделать наоборот и присвоить объекту параметризованного типа объект сырого.

Пример.

```
//описание параметризованного класса PairLair и двух
//классов-маркеров Goblin и Genie такие же, как и
//раньше
public class Main
{
    public static void main(String[] args) {
        PairLair obgRaw = new PairLair();
        PairLair<Goblin, Genie> obj = obgRaw;
    }
}
```

Можно вызвать метод обобщенного класса через объект сырого типа.

Пример.

```
//описание параметризованного класса PairLair и двух
//классов-маркеров Goblin и Genie такие же, как и
//раньше
public class Main
{
    public static void main(String[] args) {
        PairLair<Goblin, Genie> obj =
                                    new PairLair<>();
        PairLair obgRaw = obj;
        obgRaw.setObj1(new Goblin());
    }
}
```

Замечание.

Отсутствие сообщений компилятора, а также успешное выполнение приведенных выше кодов свидетельствует о допустимости подобных действий.

Наследование в обобщенных типах

При наследовании от обобщенного класса класс-наследник должен передавать данные о типе в конструкции базового класса. Формат создания класса наследника от параметрического класса:

```
спецификатор class ИмяGeneric <ИмяТипа1,…, ИмяТипаN>
                  extends СуперТип<ИмяТипа1,…, ИмяТипаN> {
    ... // тело класса
}
```

Пример.

```java
class Account<T> {
    private T id;
    //методы
    T getId(){return id;}
    Account(T id) {
        this.id = id;
    }
}

class DepositAccount<T> extends Account<T> {
    DepositAccount(T id){
    //обращение к конструктору базов класса super()
        super(id);
    }
}

public class Main
{
    public static void main(String[] args) {
        DepositAccount a =
                        new DepositAccount(20);
        System.out.println(a.getId());
        DepositAccount b =
                     new DepositAccount("12345");
        System.out.println(b.getId());
    }
}
```

Результат выполнения программы:

```
20
12345
```

В конструкторе `DepositAccount()` идет обращение к конструктору базового класса, в который передаются данные о типе.

Обращает на себя внимание то, что угловые скобки при создании объектов опущены, а компилятор сам выводит тип из контекста, т.е. из типа передаваемого значения.

Класс-наследник может добавлять и использовать какие-то свои параметры типов.

Пример.

```
1  class Account<T> {
2      private T id;
3      //методы
4      T getId(){return id;}
5      Account(T id) {
6          this.id = id;
7      }
8  }
9
10 class DepositAccount<T, S> extends Account<T>{
11     private S name;
12     //методы
13     S getName() { return name; }
14     DepositAccount(T id, S name){
15         super(id);
16         this.name=name;
17     }
18 }
19
20 public class Main
21 {
22     public static void main(String[] args) {
23         DepositAccount<Integer, String> a =
24                     new DepositAccount(20, "Tom");
25         System.out.println(a.getId() + " : " +
26                                     a.getName());
27         DepositAccount<String, Integer> b =
28                 new DepositAccount("12345", 23456);
```

```
29            System.out.println(b.getId() + " : " + 
30                                               b.getName());
31        }
32    }
```

Результат работы программы:

```
20 : Tom
12345 : 23456
```

Класс-наследник вообще может не быть обобщенным. В этом случае при наследовании явным образом указывается тип, который будет использоваться конструкциями базового класса (в примере ниже - это тип `Integer`). Необходимо следить, чтобы в конструктор базового класса передавалось именно значение указанного явным образом типа (в примере ниже – это целое число 10).

Пример.

```
1   class Account<T> {
2       private T id;
3       //методы
4       T getId() { return id; }
5       Account(T id) {
6           this.id = id;
7       }
8   }
9   
10  class DepositAccount extends Account<Integer> {
11      //методы
12      DepositAccount(Integer i) {
13          super(i);
14      }
15  }
16  
17  public class Main
18  {
19      public static void main(String[] args) {
20          DepositAccount a = new DepositAccount(10);
21          System.out.println(a.getId());
22      }
23  }
```

Результат работы программы:

```
10
```

Также может быть ситуация, когда только наследник является *параметризованным* классом, а базовый класс - обычным необобщенным. В этом случае использование конструкций базового класса в наследнике происходит как обычно.

Пример.

```
1  class Account {
2      private String name;
3      //методы
4      String getName(){return name;}
5      Account(String name)
6      {
7          this.name=name;
8      }
9  }
10
11 class DepositAccount<T> extends Account {
12     private T id;
13     //методы
14     T getId() { return id; }
15     DepositAccount(String name, T id) {
16         super(name);
17         this.id = id;
18     }
19 }
20 public class Main
21 {
22     public static void main(String[] args) {
23         DepositAccount a =
24                     new DepositAccount("Tom", 10);
25         System.out.println(a.getName() +
26                                " " + a.getId());
27     }
28 }
```

Результат работы программы:

```
Tom 10
```

Рассмотрим восходящее преобразование обобщенных типов. Как и при наследовании обычных классов объект класса наследника преобразуется «по умолчанию» к объекту класса-предка. Если вернуться к первому примеру этого пункта, то можно привести объект `DepositAccount<Integer>` к `Account<Integer>` или `DepositAccount<String>` к `Account<String>`.

Пример.

```
1  class Account<T> {
2      private T id;
3      //методы
4      T getId(){return id;}
5      Account(T id) {
6          this.id = id;
7      }
8  }
9
10 class DepositAccount<T> extends Account<T> {
11     DepositAccount(T id){
12         super(id);
13     }
14 }
15
16 public class Main
17 {
18     public static void main(String[] args) {
19         Account a = new DepositAccount(20);
20         System.out.println(a.getId());
21         //явное преобразование параметризов. типов
22         //избыточный, но верный синтаксис
23         Account<Integer> b = new DepositAccount(100);
24         System.out.println(b.getId());
25
26         Account c = new DepositAccount("12345");
27         System.out.println(c.getId());
28     }
29 }
```

Результат выполнения программы:

```
20
100
12345
```

Замечание.

Сделать то же с объектами предка и наследника, созданными с параметрами, имеющими разные значения типа нельзя.

Обобщенные типы в качестве ограничений

В качестве ограничений при задании параметров класса также могут использоваться параметризованные типы. Т.е. если параметризованный класс предок имеет множество классов наследников, то при параметризации вновь создаваемого класса допустимо использовать конструкцию:

```
<T extends КлассПредок<Тип>>
```

Пример.

```
public class MyClass <S extends Account<Integer>>
{
    public static void main(String[] args) {
    }
}
//иерархия классов-маркеров
class Account<T> { }
class DepositAccount<T> extends Account<T> { }
```

Преимущества использования параметризованных классов

Кратко перечислим преимущества:

- избежание повторяемости написания программного кода для разных типов данных. Программный код (классы и методы) пишется для некоторого обобщенного типа `T`;
- уменьшение текстовой части программного кода, и, как следствие, повышение читаемости программ;
- обеспечение удобного механизма передачи аргументов в параметризованный класса с целью их обработки методами класса.

Пример класса демонстрирующий, что параметризация не всегда нужна

В некоторых примерах, приведенных выше иногда использовалась конструкция <T extends Number>, что как известно, указывает на то, что класс, являющийся фактическим значением параметра T должен расширять класс Number.

Однако зная о том, что объекты всех числовых классов-оберток приводятся к Number автоматически, то в этом случае абсолютно необязательно использовать параметризацию для того, чтобы создать класс MyArray, работающий с числовыми массивами произвольного типа (в смысле типов классов-оберток).

Пример.

```
1   import java.util.Arrays;
2
3   class MyArray {
4       private Number[] array;
5       //конструкторы
6       public MyArray(Number[] a) {
7           this.array = Arrays.copyOf(a, a.length);
8       }
9       public MyArray(int n, int MIN, int MAX) {
10          this.array = new Number[n];
11          for(int i = 0; i < n; i++) {
12              array[i] = ((Double)(Math.random() *
13                      (MAX - MIN) + MIN)).intValue();
14          }
15      }
16      //экземплярные методы
17      public Number[] getData() {
18          return array;
19      }
20      public void setData(Number s, int i) {
21          array[i] = s;
22      }
23      public String toString() {
24          return Arrays.toString(array);
25      }
26  }
27
```

```
28  public class Main
29  {
30      public static void main(String[] args) {
31          MyArray a = new MyArray(5, 0, 100);
32          System.out.println(a.toString());
33
34          Double[] b = {1., 2., 3., 4., 5.};
35          MyArray array = new MyArray(b);
36          System.out.println(array.toString());
37
38          array.setData(6., 0);
39          System.out.println(array.getData()[0]);
40      }
41  }
```

Результат работы программы:

```
[9, 51, 21, 20, 99]
[1.0, 2.0, 3.0, 4.0, 5.0]
6.0
```

Следует отметить, что при использовании параметризации (например, с одним параметром `T`) создать второй конструктор, заполняющий случайными числами одномерный массив невозможно из-за невозможности приведения типа значений `double`, возвращаемого методом `random()` к параметризованному типу `T`.

Далее, внутри этого конструктора при использовании параметризации невозможно будет выделить память под хранение массива длинной `n` элементов (т.е. невозможно использовать инструкцию `this.array = new T[n];`).

Более того использование приведения типов выглядит более надежно и профессионально, чем тяжеловесные конструкции с параметрами.

Параметризация интерфейса

Формат объявления параметризованного интерфейса `ИмяGeneric`:

```
специф interface ИмяGeneric <ИмяТипа1,…, ИмяТипаN> {
    ... // тело интерфейса
}
```

где ~~специф~~ – спецификатор доступа; ИмяТипа1, …, ИмяТипаN – обобщенные имена типов (свободно выбираемые идентификаторы), которые используются методами интерфейса; ИмяGeneric – имя параметризованного интерфейса.

Замечания:

- *в списке перечисления параметров интерфейса допустимо использовать ключевые слова extends/super в уже известных синтаксических конструкциях: ИмяТипа extends Тип или ИмяТипа super Тип;*
- *ключевые слова extends/super могут применяться в произвольном числе параметров от общего их числа, использованных при параметризации интерфейса.*

Пример.

Main.java | AnimalBehavior.java

```
1 interface AnimalBehavior <T, V extends Number> {
2     String behavior(T s, V q);
3 }
```

Реализация параметризованного интерфейса

Класс использует ключевое слово implements для реализации интерфейса. Формат реализации классом ИмяКласса параметризованного интерфейса ИмяИнтерфейсаGen:

Существует два способа реализации интерфейса:

- реализация параметризованного интерфейса в ***не***параметризованном классе.

```
спец class ИмяКласса implements ИнтерфейсДжен<Параметры>{
   // описание класса
}
```

В этом случае список Параметры должен содержать конкретные названия типов, используемых в интерфейсе.

Однако, методы класса, реализующие параметризованный интерфейс должны иметь в качестве типов возвращаемых значений и получаемых ими объектов названия конкретных классов.

При реализации интерфейса в ***параметризованном*** классе, имя параметра класса должно совпадать с именем параметра интерфейса:

```
спец class КлассGen <Праметры1> implements
                                   ИмяИнтерфейсGen<Праметры2> {
   // описание класса
}
```

В этом случае списки `Праметры1` и `Праметры2` должны содержать имена формальных параметров, используемых как в классе, так и в интерфейсе. Число формальных параметров списка `Праметры1` класса, реализующего интерфейс, должно быть не меньше числа формальных параметров списка `Праметры2` интерфейса. Другой случай (наоборот число параметров в `Праметры1` меньше чем в `Праметры2`) будет рассмотрен ниже.

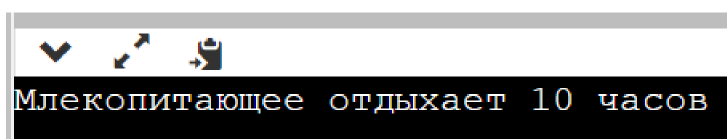
Пример.

```
// Файл AnimalBehavior.java определен выше
```

Main.java | AnimalBehavior.java

```
1  public class Main <T, V extends Number>
2                    implements AnimalBehavior<T, V> {
3      @Override
4      public String behavior(T s, V q) {
5          return "Млекопитающее " + s.toString() +
6                 " " + q.toString() + " часов";
7      }
8
9    public static void main(String args[]) {
10     Main<String, Integer> m = new Main<>();
11     System.out.println(m.behavior("отдыхает", 10));
12   }
13 }
```

Результат работы программы:



Реализация параметризованным классом интерфейса с большим числом параметров чем у класса

Если по каким-либо причинам список формальных параметров интерфейса будет иметь большее число формальных параметров, то лишние параметры (по отношению к списку параметров класса) должны быть заняты конкретными наименованиями классов.

Продемонстрируем, каким образом обойти этот случай. Рассмотрим *формальный* пример, когда у интерфейса два параметра, а у класса, реализующего этот интерфейс, один.

Пример.

```
// Файл AnimalBehavior.java определен выше
```

Main.java | AnimalBehavior.java

```
1  public class Main <T> implements
2                           AnimalBehavior<T, Integer> {
3      @Override
4      public String behavior(T s, Integer q) {
5          return "Млекопитающее " + s.toString() +
6                 " " + q.toString() + " часов";
7      }
8
9    public static void main(String args[]) {
10     Main<String> m = new Main<>();
11     System.out.println(m.behavior("отдыхает", 10));
12   }
13 }
```

Результат работы программы уже известен и демонстрировался ранее.

Возможна реализация *параметризованного* интерфейса даже обычным классом без параметров. В этом случае параметры типов в интерфейсе должны быть заменены на конкретное значение типов (классов), а параметры переопределяемых методов в классе реализующим интерфейс должны иметь соответствующие значения параметров.

В частности, определим класс `Main` из предыдущего примера *без параметров*, хотя он будет реализовывать известный уже *параметризованный* интерфейс `AnimalBehavior`.

Пример.

```
// Файл AnimalBehavior.java определен выше
```

Main.java | AnimalBehavior.java

```
1  public class Main implements
2                      AnimalBehavior<String, Integer> {
3      @Override
4      public String behavior(String s, Integer q) {
5          return "Млекопитающее " + s.toString() +
6                 " " + q.toString() + " часов";
7      }
8
9    public static void main(String args[]) {
10     Main m = new Main();
11     System.out.println(m.behavior("отдыхает", 10));
12   }
13 }
```

Замечание.

В обоих последних примерах компилятором Java считается, что методы с заголовками `public String behavior(T s, Integer q)` *и* `public String behavior(String s, Integer q)` *переопределяют абстрактный метод параметризованного интерфейса* `String behavior(T s, V q)`*. На это указывает то, что компилятор пропускает все приведенные выше конструкции. Если бы он считал, что это перегрузка, а не переопределение, то сообщал бы, что класс* `Main` *является абстрактным.*

Доступ к реализациям объектов через ссылки на параметризованный интерфейс

Переменные можно также объявлять, как объектные ссылки, которые используют тип параметризованного интерфейса, но они как и ранее должны быть инициализированы объектами реализующими интерфейс.

Диспетчеризация кода может выполняться посредством интерфейса без необходимости наличия каких-либо сведений о «вызывающем» объекте. Этот процесс аналогичен использованию ссылки на суперкласс для доступа к объекту подкласса.

Пример.

```
// Файл AnimalBehavior.java определен выше
```

Main.java | AnimalBehavior.java

```
1  public class Main <T, V extends Number>
2                            implements AnimalBehavior<T, V> {
3      @Override
4      public String behavior(T s, V q) {
5         return "Млекопитающее " + s.toString() +
6                " " + q.toString() + " часов";
7      }
8  
9    public static void main(String args[]) {
10     AnimalBehavior<String, Integer> m = new Main<>();
11     System.out.println(m.behavior("отдыхает", 10));
12   }
13 }
```

Результат работы программы будет совпадать с прежним результатом.

Хотя приведенный пример формально показывает, как ссылочная переменная интерфейса может получать доступ к объекту реализации, он не демонстрирует все полиморфные возможности такой ссылки. Чтобы продемонстрировать пример такого применения, создадим две отдельные реализации интерфейса `AnimalBehavior` в классах `Tiger` и `Zebra`.

Замечание.

Формально для усиления пример содержит реализацию параметризованного интерфейса, имеющего больше формальных параметров чем у реализующего его класса.

Пример.

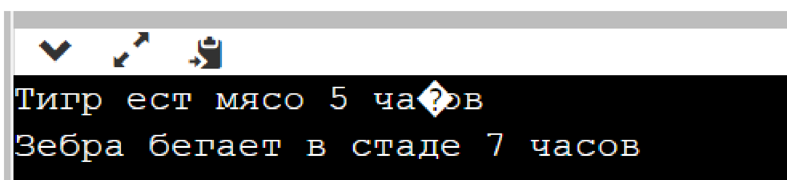
```
// Файл AnimalBehavior.java определен выше
```

Main.java | AnimalBehavior.java

```
1  class Tiger<T> implements AnimalBehavior<T, Integer>{
2      @Override
3      public String behavior(T s, Integer q) {
4          if(s.equals("ест")) {
5              return "Тигр " + s.toString() +
6             " мясо " + q.toString() + " часов";
7          }
8          else if(s.equals("ходит")) {
```

```
 9              return "Тигр " + s.toString() +
10                     " поодиночке " + q.toString() +
11                      " часов";
12          }
13          return "не определено";
14      }
15  }
16
17  class Zebra<T, V extends Number> implements
18                               AnimalBehavior<T, V> {
19      @Override
20      public String behavior(T s, V q) {
21          if(s.equals("ест")) {
22              return "Зебра " + s.toString() +
23                     " траву "+ q.toString() + " часов";
24          }
25          else if(s.equals("бегает")) {
26              return "Зебра " + s.toString() +
27                     " в стаде " + q.toString() +
28                      " часов";
29          }
30          return "не определено";
31      }
32  }
33
34  public class Main {
35      public static void main(String args[]) {
36          AnimalBehavior<String, Integer> m =
37                                   new Tiger<>();
38          System.out.println(m.behavior("ест", 5));
39          m = new Zebra<>();
40          System.out.println(m.behavior("бегает", 7));
41      }
42  }
```

Результат работы программы:


```
Тигр ест мясо 5 ча�ов
Зебра бегает в стаде 7 часов
```

Замечания:

- *дженерики-интерфейсы работают только с объектами, поэтому в качестве значений параметров типа должны использоваться только классы или если необходимо использовать базовые типы, то их классы-обертки;*
- *соглашение о наименовании параметров у интерфейса полностью повторяет соглашение о наименовании параметров классов.*

Вложенные (внутренние) параметризованные интерфейсы

Формат объявления в *параметризованном* классе **КлДжен** вложенного (внутреннего) параметризованного интерфейса **ВнИнтерфДжен**:

```
public class КлДжен <Параметры1> {
    public interface ВнИнтерфДжен <Параметры2> {
        // описание методов, составляющих интерфейс
    }
    // необязательно: свойства и методы класса
}
```

Формат реализации вложенного параметризованного интерфейса **КлДжен.ВнИнтерфДжен** внешним классом **ИмяКласса**:

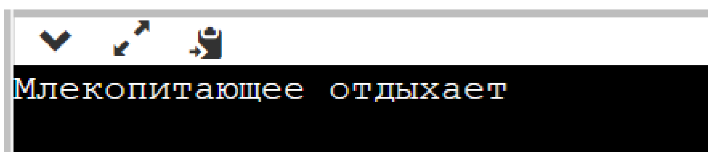
```
public class ИмяКласса<Параметры1> implements
                            ИмяКл.ИмяВнИнтерф<Параметры2> {
    //определение класса с переопределенными методами
    //из вложенного интерфейса КлДжен.ВнИнтерфДжен
    }
}
```

Пример.

```
1 class A <T> {
2     //внутренний параметризованный интерфейс в
3     //параметризованном классе
4     interface Behavior <T> {
5         String behavior(T s);
6     }
7 }
```

```
 8
 9 class Animal <T> implements A.Behavior<T> {
10     @Override
11     public String behavior(T s) {
12         return "Млекопитающее " + s.toString();
13     }
14 }
15
16 public class Main <T>
17 {
18     public static void main(String args[]) {
19        A.Behavior<String> m = new Animal<>();
20        System.out.println(m.behavior("отдыхает"));
21     }
22 }
```

Результат работы программы:



Ранее приведен пример *параметризованного* интерфейса, вложенного в *параметризованный* класс, но он может также быть вложенным в другой *параметризованный* интерфейс.

Пример.

```
interface A <T> {
    //внутренний параметризованный интерфейс в
    //параметризованном интерфейсе
    interface Behavior <T> {
        String behavior(T s);
    }
}
```

Если в предыдущем примере заменить **class A <T>** на **interface A <T>**, то результат работы программы не измениться.

Количество параметров у внешнего класса, как и внешнего интерфейса может отличаться от количества параметров у внутреннего интерфейса:

Самым очевидным случаем является случай, когда количество параметров внешнего класса или интерфейса больше количества параметров вложенного интерфейса. В этом случае вообще не требуется каких-либо действий от разработчика. Единственное, в чем он должен быть уверен так это в том, что внешний класс или интерфейс используют все параметры (т.е. они необходимы).

Во втором случае, когда количество параметров внешнего класса или интерфейса больше количества параметров внутреннего интерфейса, как уже следует из изложения следует использовать типы-«заглушки» вместо лишних параметров.

Расширение параметризованных интерфейсов

Параметризованный интерфейс может расширить другой *параметризованный* интерфейс также как и без *параметризации*. Это выполняется с помощью ключевого слова `extends`. Формат расширения интерфейсом `РасшИнтерфGen` другого интерфейса `БазИнтерфGen`:

```
public interface РасшИнтерфДжен
                                extends БазИнтерфДжен {
    // перечисление дополнительных методов к методам
    // базового параметризованного интерфейса
}
```

Один *параметризованный* интерфейс, в отличие от классов, может расширять несколько *параметризованных* интерфейсов. В этом случае интерфейсы-предки перечисляются через запятую после слова `extends`. Формат множественного расширения *параметризованных* интерфейсов:

```
public interface РасшИнтерфДжен extends
                           Список, Баз, БазИнтерфДжен {
    // перечисление дополнительных методов к методам
    // из списка базовых интерфейса
}
```

Рассмотрим пример *параметризованного* интерфейса `Sport` расширяющего *параметризованные* интерфейсы `Football` и `Hockey`. В свою очередь *параметризованный* интерфейс `TVProgram` расширяет интерфейс `Sport`. Класс `Main`, реализующий интерфейс `TVProgram`, должен переопределить методы всех интерфейсов `TVProgram`, `Sport`, `Football` и `Sport`.

Пример.

```java
interface Hockey<P> {
    String goalStatus(P r);
    String overtimePeriod(P r);
}

interface Football<S> {
    String goalStatus(S r);
    String additionTime(S r);
}

interface Sport <T> extends Football<T>, Hockey<T> {
    String setHomeTeam(T r);
    String setVisitingTeam(T r);
}

//расширение одного интерфейса
interface TVProgram<T, S> extends Sport <S> {
    String switchToChannel(T r);
}

//реализация расширенного интерфейса в классе
public class Main<T, S> implements TVProgram<T, S> {
    @Override
    public String goalStatus(S r){
        return "Гол " + r.toString();
    }

    @Override
    public String overtimePeriod(S r){
        return "Идет " + r.toString();
    }

    @Override
    public String additionTime(S r){
        return "Идет " + r.toString();
    }

    @Override
```

```
    public String setHomeTeam(S r) {
        return "Домашняя команда " + r.toString();
    }

    @Override
    public String setVisitingTeam(S r) {
        return "Приезжая команда " + r.toString();
    }

    @Override
    public String switchToChannel(T r){
        return "Включен канал " + r.toString();
    }

  public static void main(String[] args) {
    TVProgram<Integer, String> p = new Main<>();
    System.out.println(p.switchToChannel(17));
  }
}
```

Результаты работы программы:

Замечание.

Расширяющий интерфейс, очевидно может иметь больше параметров чем расширяемые (см. пример), но в случае, когда некоторые параметры расширяемых интерфейсов не нужны, то как и ранее следует использовать параметры-заглушки из созданных классов.

Реализация множества параметризованных интерфейсов в классах

Java предоставляет возможность реализации нескольких *параметризованных* интерфейсов в классе, аналогично случаю расширения множества интерфейсов. Также как и при рассмотрении предыдущей темы в случае реализации множества *параметризованных* интерфейсов в рамках одного класса все они перечисляются через запятую после слова `implements`.

Условно можно разделить реализацию множества *параметризованных* интерфейсов в параметризованных и непараметризованных (обычных) классах.

Рассмотрим в начале *параметризованный* класс, реализующий множество *параметризованных* интерфейсов.

Пример.

```
interface Printable<P> {
    // методы интерфейса
}
interface Searchable<S> {
    // методы интерфейса
}

class Book<T, V> implements Printable<T>, Searchable<V> {
    // реализация параметризованного класса Book
}
```

Реализацию множества *параметризованных* интерфейсов в обычном классе без *параметров* можно очевидно осуществить с помощью задания конкретных типов для параметров при реализации *параметризованных* интерфейсов.

Пример.

```
interface Printable<P> {
    // методы интерфейса
}
interface Searchable<S> {
    // методы интерфейса
}

class Book implements
                Printable<String>, Searchable<Integer> {
    // реализация обычного класса Book без параметров
}
```

Дефолтные параметризованные методы

В *параметризованном* интерфейсе допускается *параметризация* дефолтных методов.

Пример.

```
interface Printable<T> {
    default<T> String print(T t){
        return "Количество " + t.toString();
    }
}

class Journal<T> implements Printable<T> {
    private String name;
    // методы
    Journal(String name){
        this.name = name;
    }
    String getName(){
       return name;
    }
}

public class Main {
  public static void main(String[] args) {
    Printable<Integer> p =
                 new Journal<> ("Экономист Беларуси");
    System.out.println(p.print(100));
  }
}
```

Результат работы программы:

Параметризованные статические методы

В *параметризованных* интерфейсах доступны статические методы – это методы интерфейса, перед которыми написан спецификатор `static`. В *параметризованных* интерфейсах статические методы, как и дефолтные (методы «по умолчанию») должны быть представлены вместе с реализациями.

Таким образом, в отличие от абстрактных методов, представленных только заголовком, статические методы имеют тело, а также их не возможно переопределить в классе, реализующем интерфейс.

Замечание.

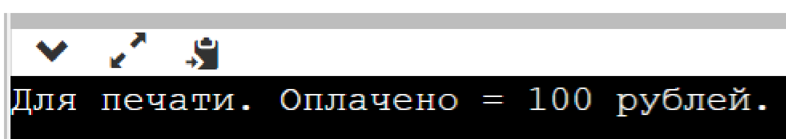
Существует общее правило для параметризованных или ***не****параметризованных статических методов: любой статический метод невозможно переопределить, но возможно перегрузить (если изменить типы или количество формальных параметров).*

Для того чтобы обратиться к статическому методу *параметризованного* интерфейса также, как и в случае со статическими методами обычных классов, пишут название интерфейса и метод: ИмяИнтерфейса.ИмяСтатичМетода(~~парам~~).

Пример.

```
1  interface Printable<T> {
2      String print();
3      // статический метод интерфейса
4      static <T extends Number> String read(T t){
5          return "Для печати. Оплачено = " +
6                  t.toString() + " рублей.";
7      }
8  }
9
10 public class Main<T> implements Printable<T> {
11     @Override
12     public String print() {
13        return "Неопределено";
14     }
15     // статический метод не переопределяется
16     public static void main(String[] args) {
17        //для вызова метода нет необходимости в объекте
18        System.out.println(Printable.read(100));
19     }
20 }
```

Результат работы программы:



Приведем пример еще одного *параметризованного* статического метода в *параметризованном* интерфейсе, возвращающего значение.

Пример.

```
1  interface Arithmetic
2                 <T extends Number, S extends Number> {
3  //статический метод, выражение extends Number в
4  //заголовке использовать обязательно иначе не найдет
5  //переопределенных методов doubleValue() для разных
6  //классов-оберток
7      static <T extends Number, S extends Number>
8                            Double multiply(T a, S b) {
9          return a.doubleValue() * b.doubleValue();
10     }
11 }
12
13 public class Main<T extends Number, S extends Number>
14                           implements Arithmetic<T, S> {
15    public static void main(String[] args) {
16       System.out.println(Arithmetic.multiply(2, 3.));
17    }
18 }
```

Результат работы программы:

Приватные параметризованные методы

По умолчанию все методы в том числе и *параметризованные* в *параметризованном интерфейсе* имеют модификатор `public`. Однако в настоящее время можно определять в интерфейсе параметризованные методы с модификатором `private`.

Подобные *параметризованные* методы, как и их обычные (***не****параметризованные*) варианты могут использоваться только внутри самого интерфейса, в котором они определены.

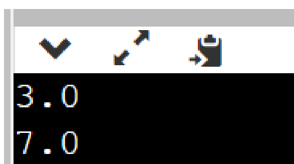
Пример.

```
interface Summable<T extends Number> {
    default Double sum(T a, T b){
        return result(a, b);
    }
    default Double sum(T a, T b, T c){
        return result(a, b, c);
    }
    //приватный метод с переменным числом аргументов
    private Double result(T... values){
        Double result = 0.;
        for(T n : values){
            result += n.doubleValue();
        }
        return result;
    }
}

class Calculation<T extends Number>
                        implements Summable<T> {
}

public class Main {
    public static void main(String[] args) {
        Summable<Integer> r = new Calculation<>();
        System.out.println(r.sum(1, 2));
        System.out.println(r.sum(1, 2, 4));
    }
}
```

Результат работы программы:



Замечание.

*Параметризованные интерфейсы, как и обычные (**не**парвметризованные) могут использоваться в качестве типа параметров метода или в качестве типа возвращаемого им значения. При*

этом конкретно получаемыми и возвращаемым значением должны быть объекты параметризованных классов, реализующих определенный параметризованный интерфейс.

К вопросу о приведении типов при реализации обобщений

При использовании обобщенных классов виртуальная машина ничего не знает об обобщенных классах. Она работает с обычными классами. На этапе компиляции производится преобразование обобщенного класса в класс, в котором все обобщенные типы заменяются либо классом `Object`, либо классом или интерфейсом, указанным после ключевого слова `extends`. При этом компилятор в нужных местах вставляет приведение типа к реальному типу, который указывается при создании экземпляра класса.

Поэтому, если не указано ограничений на тип, то компилятор может пропустить работу с переменными, имеющими обобщенный тип, только в рамках перегруженных методов типа `Object`, например, `toString()`.

Когда накладывается ограничение классом, то становятся доступными методы этого класса, и обобщенный тип будет иметь тип ограничивающего класса. Если наложено ограничение интерфейсом, то обобщенный тип будет иметь тип интерфейса, и можно получить доступ к методам, объявленным в интерфейсе.

Самым существенным минусом использования параметризованных интерфейсов является то, что даже в случае использования одного параметра, например, `T`, он должен иметь возможность приводится к классам оберткам или другим типам.

В рассмотренных выше примерах для решения этой проблемы широко использовались методы `toString()` или метод `doubleValue()` если параметр расширял класс `Number`.

К сожалению, общей методики приведения обобщенного типа `T` к произвольному типу (кроме объекта `String` методом `toString()`, определенному в суперклассе `Object`) не существует.

Любое подобное приведение типов обобщенных параметров друг к другу или другому заданному типу будет требовать разработки алгоритма, основанного на знании точной структуры конкретного класса, приходящего в качестве значения обобщенного параметра.

Именно эта конкретизация при определении метода преобразования типов друг к другу не просто значительно снижает значение параметризации, а практически уничтожает ее.

Поэтому прежде чем использовать параметризацию следует создать базовый супертип (по аналогии с Number) реализующий базовый интерфейс (абстрактные методы приведения Number к любому из наследников) и разветвленную систему производных типов, каждый из которых будет приводится к базовому типу (например, Number) «по умолчанию» (восходящее приведение типов).

В этом случае использование значений обобщенных параметров будет ограничено системой производных типов, а использование ключевого слова extends, ограничивающего значение параметров только производными типами будет гарантировать возможность беспрепятственного использования переопределенных методов супертипа в том числе для приведения.

Однако при подобном использовании наследования необходимость параметризации может отпасть сама собой, как это было продемонстрировано в примере разработанного ранее класса, который в качестве свойств имел массив суперкласса Number и этим обеспечивал универсальность вычислительной обработки массива любых числовых типов (производных от Number) без использования параметризации.

Дополнительные сведения по обобщениям

Выведение типов

Выведение типов — это способность компилятора Java автоматически определять аргументы типа на основе контекста, чтобы вызов получился возможным. Алгоритм выведения типов определяет типы аргументов и, если есть, тип, в который присваивается результат или в котором возвращается результат. Далее алгоритм пытается найти наиболее подходящий тип, который работает со всеми аргументами.

Целевые типы

Компилятор Java пользуется целевыми типами для вывода параметров типа при вызове обобщенного метода. Целевой тип выражения — это тип данных, который компилятор Java ожидает в зависимости от того, в каком месте находится выражение. Рассмотрим экземпляр класса Collections и вызов с его помощью экземплярного метода emptyList() (Collections.emptyList()), который имеет следующий заголовок:

```
static <T> List<T> emptyList()
```

Рассмотрим следующую инструкцию присвоения с использованием этого метода:

```
List<String> listOne = Collections.emptyList();
```

Эта инструкция ожидает возвращение методом `emptyList()` экземпляра `List<String>`. Этот тип данных является целевым типом. Поскольку метод `emptyList` возвращает значение типа `List<T>`, то компилятор выводит, что аргумент типа `T` будет типом `String`.

Стирание типа (Type Erasure)

На самом деле речь пойдет о стирании ***параметров*** обобщенных типов, но так исторически установился сленг.

Для реализации любых обобщений компилятор Java применяет стирание типа (`type erasure`). Во время процесса стирания типов компилятор Java стирает все ***параметры*** типа и заменяет каждый его ограничением, если параметр типа ограничен, либо `Object`-ом, если параметр типа неограничен.

Компилятор Java также стирает параметры типа обобщенных методов и интефейсов.

Литература

1. Блинов, И.Н. Java from EPAM / И.Н. Блинов, В.С. Романчик. - Минск: Четыре четверти, 2020. - 560 с.
2. Учебники по Java - [Электронный ресурс], URL: https://docs.oracle.com/javase/tutorial/java/generics/index.html (дата обращения: 08.06.22)
3. Руководство по языку программирования Java/ METANIT.COM - [Электронный ресурс], URL: https://metanit.com/java/tutorial/ (дата обращения: 08.06.22)
4. JavaRush - [Электронный ресурс], URL: https://javarush.ru/ (дата обращения: 08.06.22)

Учебное издание

Кравчук Александр Степанович
Кравчук Анжелика Ивановна
Кремень Елена Васильевна

Язык Java.
Дженерики

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

В авторской редакции

Ответственный за выпуск *Е. В. Кремень*

Подписано в печать 20.02.2023. Формат 60×84/16. Бумага офсетная.
Усл. печ. л. 3,02. Уч.- изд. л. 2,91. Тираж 50 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 1/270 от 03.04.2014.
Пр. Независимости 4, 220030, Минск.

Отпечатано с оригинал-макета заказчика
на копировально-множительной технике
механико-математического факультета
Белорусского государственного университета.
Пр. Независимости 4, 220030, Минск.