УДК 004.422.6, 004.43

Сафонов Владимир Олегович,
Сафонова Адель Наркисовна

ПАРАМЕТРИЗОВАННЫЕ ТИПЫ (GENERICS): ИСТОРИЯ, ТЕОРИЯ, РЕАЛИЗАЦИЯ, ПРИМЕНЕНИЕ, ПРЕДЛОЖЕНИЯ ПО РАСШИРЕНИЮ JAVA-ТЕХНОЛОГИИ

Аннотация

В статье рассмотрена концепция параметризованных типов данных (generics), играющая ключевую роль в современном программировании, и сформулированы предложения авторов по расширению параметризованных типов в Java. Статья написана по материалам доклада на международной конференции EclipseCon 2011 Europe, Людвигсбург, Германия, ноябрь 2011 г.

Ключевые слова: параметризованные типы данных (generics), абстрактные типы данных, шаблоны, переменные виды (модалы), параметризованные пакеты, Simula 67, Ada, Java, C++, C#.

ВВЕДЕНИЕ

Параметризованный тип данных (generic type) – одна из ключевых концепций современного программирования. Параметризованным называется тип, описывающий сложную структуру данных (*коллекцию*), состоящую из однотипных элементов – например стек, дерево, граф, очередь, хеш-таблица, – в котором тип элементов является *параметром*, например *Stack <t>* (в синтаксисе языка Java). Алгоритмы обработки коллекций – обход элементов, генерация элементов, поиск и др. – носят общий характер и не зависят от типа элемента (параметра), либо зависят от него некоторым, явно специфицированным образом – используют определенные *операции*, требуемые от фактического параметра-типа (например операцию линейного упорядочения в обобщенном алгоритме сортировки массива). В связи с общностью самой концепции и алгоритмов обработки параметризованных типов, в настоящее время распространен другой русский перевод термина *generics – обобщенные типы данных*, который считаем вполне уместным, хотя все же предпочитаем использовать термин *параметризованные типы.*

Важная цель данной статьи – дать молодым программистам более полное представление о концепции параметризованного типа, что, к сожалению, не сделано во многих современных работах. Ныне подчас принято поверхностно излагать данную концепцию в терминах конкретного языка (например Java или C#), в то время как исторические корни ее лежат в работах 1960–1970-х гг., именно из них заимствованы, да и то не в полной мере, возможности па-

раметризованных типов в современных языках.

С точки зрения теории, концепция параметризованного типа данных близка другой концепции – *абстрактного типа данных (АТД)*, в англоязычной терминологии – *abstract data type (ADT)*. Концепция АТД была впервые сформулирована Ч. Хоаром (C.A.R. Hoare, Оксфордский университет) в 1972 г. в статье [1]. Другой основоположник данной концепции – Б. Лисков (B. Liskov, Массачусетский технологический институт), автор языка CLU [2], разработанного в 1974 г. и основанного на концепции АТД, лауреат Премии А. Тьюринга 2009 года.

Абстрактный тип данных – это совокупность следующих элементов:

– *имени* типа;

– его *конкретного представления* (в терминах других типов), скрытого внутри определения АТД;

– *сигнатуры* – набора *абстрактных операций*, с помощью которых возможна обработка данных этого типа;

– *формальной спецификации* поведения типа и его операций. Данный элемент иногда отсутствует (например, в языке CLU); в этом случае типы принято называть *инкапсулированными*.

Очевидно, что концепция АТД наиболее удобна именно для определения коллекций, как и концепция параметризованного типа. С этой точки зрения, механизм *generics* можно рассматривать как средство определения и обработки *параметризованных абстрактных типов данных*.

Листинг 1. Параметризованный класс в языке Симула-67

```
class scanner (X); ref (tree) X;
  begin integer current;
  procedure traverse (X); ref (tree) X;
    if X =/= none then
    begin
        traverse (X. left);
        current: = X. val;
        detach;
        traverse (X. right)
    end traverse;
  traverse (X);
  current: = integer max
end scanner;
sc :- new scanner(my_tree);
```

Один из авторов статьи проф. В.О. Сафонов – специалист по абстрактным типам данных, в частности, еще в 1983–1984 гг. выступал с докладами о реализации абстрактных типов на Всесоюзных конференциях в Новосибирске (Академгородке) и в Кишиневе, вызвавшими большой интерес специалистов. В 1983–1985 гг. был руководителем группы разработчиков первой в СССР реализации языка CLU, основанного на концепции абстрактных типов данных [3, 4].

1. ИСТОРИЯ ПАРАМЕТРИЗОВАННЫХ ТИПОВ

Из наиболее широко известных языков программирования, впервые параметризованные типы (классы) появились, по-видимому, в языке Симула-67 [5]. Данный язык, предназначенный для моделирования систем с дискретными событиями, известен в истории ИТ как один из первых языков объектно-ориентированного программирования. Одной из возможностей языка Симула-67 являются *параметризованные классы*. Пример кода параметризованного класса приведен в примере 1 (листинг 1).

Из примера видно, что классы в языке Симула-67 параметризуются *объектами и значениями*, а *не типами*. В примере класс *scanner* параметризован объектной ссылкой типа *ref (tree)* на дерево, которое он обрабатывает.

Следующим широко известным языком, в который предлагалось ввести параметризованные *виды*, был Алгол 68. *Вид*, в англоязычном варианте – *mode*, – аналог термина *тип*, использованный в описании Алгола 68. Предложение под названием *modals (модалы,* или *переменные виды*) о введении параметризованных видов в Алгол-68 было сформулировано одним из авторов Алгола-68 – Ч. Линдси (C.H. Lindsey) [6]. К сожалению, данная концепция *не* была включена в окончательную стандартную версию Алгола 68.

Далее концепция параметризованного типа развивалась в языках CLU [2] и

ALPHARD [7]. Язык ALPHARD был разработан в конце 1970-х гг. М. Шо (M. Shaw) в Университете Карнеги-Меллона. Концепции языка CLU, как будет видно из дальнейшего изложения, положены в основу современных языков программирования Java и C#. Язык APLHARD стал первым языком *доказательного программирования:* описания параметризованных абстрактных типов данных в нем содержали *формальные спецификации* типа данных и его операций в стиле *исчисления программ* Р. Флойда – Ч. Хоара [8] (в виде *троек Хоара: P {S} Q*); экспериментальная реализация языка имела в своем составе *верификатор*, автоматически проверявший соответствие реализации программы на языке ALPHARD ее спецификациям. Идея авторов языка APLHARD о *языке разработки и накопления корректных реализаций алгоритмов* заслуживает особого внимания и была воспринята авторами некоторых современных систем. Например, система Spec# [9] разработки Microsoft Research представляет собой расширение языка C# средствами формальной спецификации программ в стиле *design-by-contract* по Б. Мейеру.

В конце 1970-х гг. был разработан язык Ада [10], использовавшийся затем долгое время при разработке надежного и модульного программного обеспечения для Министерства обороны США. В нем была введена конструкция *generic package – параметризованный пакет*. То есть впервые термин *generics* появился именно в языке Ада.

В настоящее время параметризованные типы распространены весьма широко и включены во многие современные языки программирования – Java, C#, F# и даже в Visual Prolog [11].

2. ПАРАМЕТРИЗОВАННЫЕ АБСТРАКТНЫЕ ТИПЫ ДАННЫХ В ЯЗЫКЕ CLU

Как описано во введении, абстрактный тип данных (АТД) в общем случае имеет следующую структуру:

ADT = <Name, Representation, Operations, Specification>,

то есть состоит из *имени*, *конкретного представления*, *сигнатуры* операций и формальной *спецификации*. Авторы данной концепции – Ч. Хоар и Б. Лисков.

Первым языком, основанным на концепции АТД, стал язык CLU (1974), разработанный проф. Б. Лисков и ее группой [2]. Название языка – аббревиатура от слова *CLUster* (основная концепция языка; определение АТД). Впервые описание языка CLU на русском языке было дано в 1989 г. в одной из глав монографии В.О. Сафонова [3].

Основные особенности и возможности языка CLU следующие.

Любой тип состоит из *конкретного представления* (обозначаемого ключевым словом ***rep***) и набора *абстрактных операций*. Другое обозначение – ***cvt*** (аббревиатура термина *ConVerT – преобразовать*), используемое при описании аргументов и результатов модулей, можно трактовать как «извне – абстрактный, изнутри – конкретный». Доступ к конкретному представлению (***rep***) извне типа невозможен: действует правило *инкапсуляции* (*encapsulation*).

Обработка объектов данного типа может быть выполнена с помощью его *абстрактных операций* $P_1, \ldots P_n$. Общая форма вызова операции типа *T*:

$$T\ \$\ P(x_1, \ldots, x_n),$$

то есть, в современной терминологии, операции типа *T* вызываются как *статические методы*.

Основные виды модулей в языке CLU: *процедура (procedure)* – абстракция исполнения; *кластер (cluster)* – абстракция данных; *итератор (iterator)* – абстракция управления. В частности, итератор – специальная конструкция, введенная впервые именно в языке CLU для организации циклов перебора (генерации) элементов сложной структуры данных (коллекции). Отметим преемственную связь итераторов в языках Java и C# с итераторами в языке CLU.

Как видим, концепция *объекта* присутствует в языке CLU. Однако его нельзя назвать *объектно-ориентированным языком*, так как в нем отсутствует наследование типов. Видимо, оно не введено по принципи-

альным соображениям, так как наследование, строго говоря, противоречит принципу инкапсуляции. В современной терминологии язык CLU – это *объектно-базированный (object-based)* язык.

В языке CLU *любая* сущность (например, целое число) является объектом. Например, *int $ add(i, k)* – целочисленное сложение, а *int* – встроенный кластер. Для более краткого инфиксного обозначения операций введено в качестве *«синтаксического сахара» (syntactic sugar)* традиционное написание $i + k$, которое возможно не только для стандартных типов, но и для типов, определенных пользователем, например, если x и y имеют пользовательский тип *Matrix*, над котором определена бинарная операция *add* в виде процедуры.

Что наиболее важно для предмета данной статьи, – в языке CLU возможно определение и использование *параметризованных* (*parametrized*) *модулей*. Параметризованными могут быть не только АТД (кластеры), но и процедуры и итераторы. Параметрами модуля могут быть не только типы, но и *константы* (например, максимальная глубина стека).

Заметим, что, несмотря на несомненную практическую полезность и простоту механизма параметров-констант, *ни в одном* из современных языков программирования (включая Java и C#), они *не реализованы*, несмотря на то, что с момента введения и реализации данной концепции в языке CLU прошло уже около 40 лет. Одно из наших предложений по расширению языка Java как раз и заключается в том, чтобы *ввести в Java параметры-константы*.

Для генерации и обработки исключиельных ситуаций в абстрактных операциях в языке CLU введены ***сигналы (signals)***. Концепция сигнала в CLU является предшественником современной концепции ***исключения (exception)***. Однако сигналы в CLU *не* являются объектами. Они определяются в заголовках модулей, в предложении ***signals*** (например *signals empty*), генерируются операторами ***signal*** (***signal empty***) и обрабатываются операторами ***except when*** (***except when empty***). Современный аналог последних – блоки ***try / catch*** (C++, Java, C#).

Все описанные базовые концепции CLU – *абстрактный тип данных (кластер), итератор, сигнал, параметризованный модуль* – или их близкие аналоги являются неотъемлемой частью современных языков и систем программирования.

Рассмотрим пример описания и использования параметризованного кластера *list (список)* на языке CLU (пример 2, листинг 2).

Дадим необходимые пояснения к примеру. Кластер *list* параметризован типом элементов списка. На тип элемента не накладывается никаких ограничений, поэтому в реализации кластера могут быть использованы только базовые возможности обработки типа *t*. К ним относятся: *описания* локальных переменных типа *t*; *присваивание* объектов типа *t* (фактически – присваивания объектных ссылок); передача объекта типа *t* в качестве аргумента, выдача в качестве результата. Как видно из описания конкретного представления (***rep***), возможно использование *массивов* типа *t*. Сравним данную возможность с языком Java: в Java, как это ни удивительно, создание массивов типа-параметра *t* (в Java он называется *типовой переменной – type variable*) *не допускается*. Причина – в специфике реализации параметризованных типов в Java (см. раздел 5). Данное ограничение возможностей весьма серьезно. В языке CLU подобного ограничения нет.

В примере определены следующие операции над списками: *create, cons, car, cdr* (процедуры) и *elems* (итератор элементов списка). Реализация каждой операции фактически базируется на аналогичной операции над типом ***rep*** – над массивами. Следует обратить особое внимание на реализацию итератора, в которой использован встроенный итератор элементов массива *elements*: при каждой итерации цикла выполняется оператор ***yield*** (*elt*), где *elt* – ссылка на текущий элемент массива (списка), который при использовании итератора типа *list* передает управление очередной итерации использующего цикла и передает ей ссылку на очередной элемент. Отметим, что в современном языке C#, в его новых версиях, введен аналогичный оператор *yield return* с такой же семантикой.

Листинг 2. Параметризованный кластер list в языке CLU и его использование

```
list = cluster [t: type] is create, cons, car, cdr, elems;
    rep = array [t];
    create = proc () returns (cvt)
      return(rep $ new());
    end create;
    cons = proc (x: t, l: cvt) returns (cvt)
        return (rep $ addl(l, x));
    end cons;
    car = proc (l: cvt) returns (t) signals (empty)
        if l = nil
        then signal empty
        else return (rep $ bottom(l));
    end car;
    cdr = proc (l: cvt) returns (cvt) signals (empty)
       if l = nil
       then signal empty
       else return (rep $ tail(l));
    end cdr;
    elems = iter (l: cvt) yields(t)
       for elt: t in rep $ elements(t) do
           yield(elt);
       end
    end elems;
end list

list_user = proc ()
    li = list [int];
    l: li := li $ cons(1, li $ cons(2, li $ create());
    for elt: int in li $ elems(l) do
        output(i)
    end
end  list_user
```

Пример описания на языке CLU параметризованной процедуры сортировки с *ограничениями* на параметр-тип:

```
sort = proc [t: type] (a: array[t])
  where t has lt: proctype(t, t) returns (bool)
        …
end sort
```

Данное ограничение (***where*** …) требует от любого фактического параметра-типа наличия бинарной операции *lt* с булевским результатом, которая и используется в теле процедуры *sort*. Подразумевается (хотя и не специфицируется явно), что это операция линейного упорядочения элементов массива.

Для сравнения, в языке Java сходное ограничение формулируется гораздо более изящно и логично, не выходя за рамки привычных для языка Java базовых концепций класса и интерфейса, например:

```
public class Stack <t implements
                                Comparable> …
```

Здесь от типа элементов параметризованного стека требуется, чтобы он реализовывал соответствующий интерфейс, содержащий операцию сравнения.

Реализация параметризованных типов данных в языке CLU. Существуют различные методы реализации параметризованных типов данных. Наиболее прямолинейным из них является *макроподстановка* – дублирование аналогичных кодов для каждого фактического параметра-типа. Данный метод приводит к резкому увеличению раз-

мера генерируемого кода. В связи с этим, авторами языка CLU рекомендован другой метод – передача в процедуры параметризованного кластера в качестве дополнительных аргументов, *указателей на процедуры*, реализующие *операции*, *требуемые* от любого фактического параметра-типа в *ограничениях*, специфицируемых в заголовке кластера. Рассмотрим следующий код на языке CLU:

```
stack = cluster <t : type> is push, pop
where t has equal: proctype (t, t)
                           returns (bool)
… t $ equal (a, b) …
end stack

si = stack <int>
s: si = si.create (); …
```

Здесь от типа-параметра *t* кластера *stack* требуется наличие операции сравнения (для элементов стека). В обозначениях данного примера, вызов операции *push* для конкретизации *si* стека типом *int*:

```
si $ push (s, 1);
```

реализуется как:

```
pointer_to_s_push
  (pointer_to_int_equals, pointer_to_s, 1);
```

где:

– *pointer_to_s_push* – указатель на код процедуры, реализующий операцию *push* типа *stack* (отметим, что этот код един для любой конкретизации стека);

– *pointer_to_int_equals* – указатель на код процедуры, реализующий операцию *equals* для встроенного типа *int*;

– *pointer_to_s* – указатель на объект конкретного типа *stack[int]*.

Данная реализация требует дополнительных действий во время выполнения, но зато позволяет избежать дублирования кода для каждой конкретизации кластера *stack*.

Возможным оппонентам подобного подхода напомним, что основой реализации нестатических методов (методов экземпляра) в *любом* языке объектно-ориентированного программирования является подобный же прием – неявное «проталкивание» ссылки на объект *this* в качестве дополнительного первого аргумента метода.

Первая отечественная реализация языка CLU. В 1985 г. группой под руководством В.О. Сафонова была разработана первая в СССР реализация языка CLU [3]. Все описанные выше возможности языка CLU в ней были реализованы, включая параметризованные модули. Более того, как выяснилось при разработке, язык CLU оказалось необходимым расширить конструкциями для отдельных определений интерфейса кластера (в понимании, близком трактовке интерфейсов, например, в современном языке Java, – как совокупности имени типа и заголовков его процедур и итераторов) и реализации кластера (как совокупности реализаций его процедур и итераторов), что и было выполнено. Из других новых идей и методов, разработанных в ходе реализации, следует отметить эффективный алгоритм обработки типов в языках со структурной идентичностью (к каковым относится язык CLU) [4].

3. ПАРАМЕТРИЗОВАННЫЕ ПАКЕТЫ (GENERIC PACKAGES) В ЯЗЫКЕ АДА

Приведем пример параметризованного пакета *Stack* в языке Ада (см. листинг 3).

В примере пакет *Stack* параметризован не только типом элемента, но и максимальным размером (глубиной) стека. Предусмотрена также генерация и обработка в необходимых случаях исключительных ситуаций переполнения и исчерпания стека.

Экземпляры параметризованного пакета *Stack* могут быть получены следующим образом:

```
package STACK_INT  is new STACK
    (SIZE => 200,   ITEM => INTEGER);
  package STACK_BOOL is new STACK
       (100, BOOLEAN);
```

Таким образом, процедуры конкретизированных пакетов могут быть вызваны следующим образом:

```
STACK_INT.PUSH(N);
STACK_BOOL.PUSH(TRUE);
```

Суммируя сказанное, отметим, что в языке Ада возможности параметризованных пакетов реализованы достаточно полно.

К сожалению, сам язык Ада в целом оказался достаточно сложен для применения

большинством программистов, а все известные его компиляторы работали очень медленно (ввиду сложности языка), что дополнительно затрудняло его использование.

4. ШАБЛОНЫ (TEMPLATES) В ЯЗЫКЕ *C++*

Рассмотрим теперь пример определения параметризованного класса *Stack* в языке C++. В нем параметризованные типы и методы называются *шаблонами (templates)* (см. листинг 4).

Шаблоны в языке C++ широко используются как полезный механизм обобщенного программирования. Широко известна, например, библиотека шаблонов *Standard Templates Library (STL)*. Однако они имеют ряд недостатков. Во-первых, шаблоны менее надежны и безопасны для использования, чем параметризованные типы в языках CLU и Java – контроль типов при использовании шаблонов выполняется лишь частично, либо вовсе отсутствует. Во-вторых, шаблоны реализуются дублированием кода. Например, компилятор с языка C++ корпорации Oracle (в прошлом – Sun) генерирует *базу данных шаблонов (templates database – Templates.DB)* в виде рабочей поддиректории, в которую записывает коды всех параметризаций шаблонов. Подобный подход приводит к значительному увеличению размера бинарного кода.

5. ПАРАМЕТРИЗОВАННЫЕ ТИПЫ В ЯЗЫКЕ JAVA

5.1. Мотивировка, цели и принципы. Параметризованные типы в Java были одним из первых официальных предложений по расширению Java-технологии – JSR 14, зарегистрированным в 1999 г. Руководителем

Листинг 3. Параметризованный пакет *Stack* на языке Ада

```
generic
   SIZE : POSITIVE;
   type ITEM is private;
package STACK is
   procedure PUSH(E : in  ITEM);
   procedure POP (E : out ITEM);
   OVERFLOW, UNDERFLOW : exception;
end STACK;

package body STACK is
    type TABLE is array (POSITIVE range <>) of ITEM;
    SPACE : TABLE(1 .. SIZE);
    INDEX : NATURAL := 0;
    procedure PUSH(E : in ITEM) is
    begin
        if INDEX >= SIZE then raise OVERFLOW;
        end if;
        INDEX := INDEX + 1;
        SPACE(INDEX) := E;
    end PUSH;
    procedure POP(E : out ITEM) is
    begin
        if INDEX = 0 then raise UNDERFLOW;
        end if;
        E := SPACE(INDEX);
        INDEX := INDEX - 1;
    end POP;
 end STACK;
```

Листинг 4. Параметризованный класс (шаблон) *Stack* на языке C++

```
template <class T>
class Stack
{
public:
      Stack(int = 10) ;
      ~Stack() { delete [] stackPtr ; }
      int push(const T&);
      int pop(T&) ;
      int isEmpty()const { return top == -1 ; }
      int isFull() const { return top == size - 1 ; }
private:
      int size ;  // number of elements on Stack.
      int top ;
      T* stackPtr ;
} ;
```

разработки спецификации был Gilad Bracha (Sun), в настоящее время им является Alex Buxley (Oracle). Предложение обсуждалось в течение 5 лет, и только в 2004 г., в рамках JDK 1.5, была разработана его эталонная реализация (Reference Implementation – RI). Цели и принципы введения параметризованных типов в Java следующие:

– введение в Java параметризованных типов данных в стиле языка CLU, адаптированных к ООП;

– эффективность и простота реализации, в частности:

– для представления параметризованных типов не должны создаваться дополнительные структуры времени выполнения;

– виртуальная машина (JVM) не должна быть усложнена обработкой параметризованных типов (выражаясь неформально, – не должна ничего знать о них)

– использование уже существующего полиморфного механизма Java – таблиц виртуальных методов (VMT), вместо передачи дополнительных аргументов параметризованным методам.

Подобный подход, по-видимому, был вызван намерением авторов Java обойтись минимальными изменениями в спецификации и реализации. Как мы увидим в дальнейшем, в нем есть и достоинства, и недостатки: с одной стороны, реализация максимально эффективна; с другой – в отличие от реализаций аналогичных механизмов в других языках, возникает эффект «стирания типов» (type erasure), в связи с тем, что в Java полностью отсутствует какое-либо явное представление параметризованных типов во время исполнения программы, то есть параметризованные типы фактически существуют только во время компиляции, а во время выполнения они «растворяются» в других базовых возможностях Java-технологии, прежде всего, в реализации виртуальных методов и их таблиц (VMT). Принадлежность к той или иной конкретизации параметризованного типа распознать в Java практически невозможно – они не подвержены анализу с помощью рефлексии.

5.2. Пример параметризованного интерфейса и его использования. В качестве простейшего примера (принадлежащего фирме Sun) рассмотрим фрагмент кода на языке Java, описывающего уже встроенные в версии Java 1.5, 1.6, 1.7 интерфейсы (см. листинг 5).

Здесь описаны два встроенных параметризованных интерфейса – *List* и *Iterator* в том виде, в каком они включены во встроенный пакет *java.util*. Подобные параметризованные интерфейсы вполне естественны, так как они специфицируют операции над *коллекциями*. Последние по своей природе являются параметризованными (типом элемента коллекции), и операции над ними носят обобщенный характер.

Использование интерфейса *List*:

```
List <String> ls = new ArrayList<String> ();
ls.add ("new string");
```

Здесь создается и инициализируется список строк.

Таким образом, описание списка параметризуется типом элемента, аналогично примеру 2 на языке CLU. Сама по себе эта идея, таким образом, далеко не новая. Однако элементы новизны имеет метод реализации параметризованных типов в языке Java. В частности, определение параметризованного интерфейса (пример 5) компилируется в *единственный* class-файл (байт-код), который, в отличие от реализации шаблонов в языке C++, *не* дублируется при каждой конкретизации.

Очевидно, что любое альтернативное решение данной задачи на языке Java – например определение аналогичных интерфейсов *ListInt, ListString, …* для реализации отдельно списков целых чисел, отдельно списков строк и т. д. – было бы недостаточно удобным и общим и привело бы к дублированию кода.

5.3. Пример параметризованного метода с ограничениями. В качестве следующего примера параметризованных типов на языке Java рассмотрим пример 6 – пример параметризованного метода *max*, выполняющего поиск в списке «максимального» элемента в смысле отношения *compareTo*, наличие которого (в виде метода с определенной сигнатурой) требуется от типа-параметра. Пример разработан фирмой Sun (см. листинг 6).

В данном примере метод *max* требует от типа-параметра *T* наличия операции сравнения в виде метода *boolean compareTo (T x)*. Это ограничение задается в виде условия *<T implements Comparable<T>>*, которое означает, что тип *T* должен реализовывать параметризованный интерфейс *Comparable<T>*, содержащий указанный заголовок метода. Операция *compareTo* используется в теле метода *max* для определения максимального элемента списка.

В общем случае подобное ограничение имеет следующий синтаксис:

Листинг 5. Параметризованные интерфейсы на языке Java

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
} // List
public interface Iterator<E> {
    E next();
    boolean hasNext();
} // Iterator
```

```
<T extends SomeClass>
```

или:

```
<T implements SomeInterface> ,
```

то есть требует, чтобы тип *T* был подклассом некоторого класса, либо реализовывал некоторый интерфейс. Допускается указание списка интерфейсов- и классов-предков:

```
<T extends C1 & … & Cn>
```

В отличие от обработки шаблонов в языке C++, компилятор Java выполняет полный статический контроль корректности параметризованных типов, в том числе – соблюдение ограничений, специфицированных в определении параметризованного класса. В этом Java-технология следует традициям языка CLU.

5.4. Различные конкретизации разделяют один и тот же class-файл. Как показывает приводимый ниже пример, анализ типов различных конкретизаций одного и того же параметризованного класса приводит к *одному и тому же* class-файлу:

Листинг 6. Параметризованный метод на языке Java

```
class ListUtilities {
public static <T implements Comparable<T>>
    T max(List<T> xs) {
    Iterator<T> xi = xs.iterator();
    T w = xi.next();
    while (xi.hasNext()) {
        T x = xi.next();
        if (w.compareTo(x) < 0) w = x;
     }
    return w;
}
```

```
Vector<String> x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
boolean b = x.getClass() == y.getClass();
```

В данном примере значение переменной *b* равно *true*, поскольку *различные* типы *Vector<String>* и *Vector<Integer>* разделяют *один и тот же* класс *Vector* во время выполнения. Этот пример показывает принципиальное различие понятий *типа* и *класса* в языке Java, применительно к параметризованным типам (во всех остальных отношениях концепции класса и типа в языке Java близки друг к другу). Данная особенность вызывает двоякое отношение. С одной стороны, такой подход позволяет избежать дублирования кода. С другой – это еще одно подтверждение того, что концепция параметризованного типа во время выполнения как бы «растворяется» в классах, что, откровенно говоря, не вполне соответствует современному подходу к типизации: в большинстве современных платформ, например, в .NET, информация о типе полностью доступна во время выполнения в виде *метаданных*.

5.5. Пример параметризованного стека и анализ его реализации. Приведем еще один пример параметризованного класса (принадлежащий фирме Sun) и проанализируем его реализацию в терминах байт-кода. Это поможет читателям в понимании нюансов реализации параметризованных типов. Пример сознательно упрощен: в нем акцентируется внимание именно на представлении и использовании информации о параметризованном типе и его методах, а все содержательные действия над стеком опущены (они заменены демонстрационной печатью) (листинг 7).

В данном примере на параметр-тип класса *Stack* накладываются следующие ограничения: он должен реализовывать метод *compareTo* (входящий в интерфейс *Comparable*) и должен быть *сериализуемым* (то есть в терминах языка Java, должен реализовывать пустой интерфейс-маркер *Serializable*).

Отметим еще раз, что *все* конкретизации класса *Stack* разделяют *один и тот же* class-файл. Однако контроль типов во время компиляции остается строгим: например, типы *Stack <Integer>* и *Stack <String>* ***не*** идентичны, поэтому присутствующее в коде примера 7 определение переменной *I* с присваиванием ей значения *s* было бы ошибкой, в случае удаления символа комментария //.

Реализация класса Stack. Компилятор Java генерирует для данного примера *два* class-файла: *Main.class* (для основного класса *Main*) и *Main$Stack.class* (для параметризованного класса *Stack,* который нас интересует в данном примере).

Листинг 7. Параметризованный класс Stack и его реализация.

```
package generics_test;
public class Main {
    public static class Stack
     <T extends Comparable<T> & java.io.Serializable> {
        public void push (T x) {
            int i = x.compareTo(x);
            System.out.println("i=" + i);
         }
        public T pop () { return null; }
    }
    public Main() {   }
    public static void main(String[] args) {
        Stack <String> s = new Stack <String>();
       // Stack <Integer> I = s; – incompatible types
        s.push(«abc»);
    }
}
```

Рассмотрим байт-код метода *main* класса *Main*, использующего параметризованный класс *Stack* (листинг 8).

Не будем вдаваться в детали спецификации байт-кода Java: его подробное описание приведено на основном сайте Java-технологии http://www.java.com. Как известно [12], байт-код основан на постфиксной записи инструкций (команд) виртуальной Java-машины JVM, а его истоки восходят к *P-коду*, разработанному для переносимых реализаций языка Паскаль в 1970-х гг. и (как и сам термин *байт-код*) к разработкам фирмы Xerox PARC (США) 1980–1990-х гг. по реализации языка Smalltalk. Команды байт-кода выполняют действия над стеком для вычисления выражений текущего метода, накапливая в нем операнды для очередной операции или аргументы для очередного вызова; после выполнения операции (вызова) операнды (аргументы) вычеркиваются из стека, а в стек помещается результат. В данном примере команда номер 0 (в качестве номеров команд приведены их относительные адреса в байтах) генерирует не инициализированный объект конкретизации класса *Stack <int>*. Возникает естественный вопрос: где передача информации о конкретном фактическом параметре *String*? Ее *нет*! В этом – особенность реализации параметризованных типов в Java. Нет ее и в команде 4, роль которой – в *инициализации* объекта конкретизации класса *Stack*. Инициализация выполняется методом *<init>*, который генерируется компилятором для каждого класса и вызывается для инициализации его объекта. Предварительно команда 3 выполняет дублирование (*dup*) ссылки на объект конкретизации класса *Stack,* которая снимается со стека вызовом системного метода *<init>* (командой 4). Результат вызова конструктора присваивается локальной переменной *s* (команда 7). Напомним, что в байт-коде локальные переменные методов не именуются, а *нумеруются*, начиная от 0 (номер 0 в статическом методе *main* зарезервирован за адресом командной строки, представленной в виде массива строк). Команды 8–11 выполняют подготовку в стеке аргументов для вызова метода *push* (собственно вызов – команда 11). Рассмотрим подробнее, как именно он выполняется. Загружается в стек объект *s*, затем – аргумент-строка *«abc»*, представленная ссылкой в *константный пул* class-файла (в константном пуле хранятся не только константы, но и информация о типах). После этого выполняется стандартным для Java образом вызов метода *push* как *виртуального* метода, код которого взят из class-файла *Main$Stack.class*. Таким образом, байт-код параметризованного класса *Stack* является самым обычным байт-кодом класса с виртуальным методом *push*. Все необходимые проверки типов выполняет компилятор: при анализе конкретизации *Stack <String>* проверяется, что класс *String* реализует интерфейс *Comparable*, то есть метод *compareTo*.

Рассмотрим теперь байт-код метода *push параметризованного класса Stack* (см. листинг 9).

Наиболее важные команды здесь – 0, 1, 2 – реализация вызова метода $x.compareTo(x)$. Данный вызов (для не известного в момент компиляции типа-параметра, которым обладает объект *x*) реализуется точно так же, как вызов данного метода у объекта *любого конкретного типа, реализующего интерфейс*

Листинг 8. Байт-код метода *main*, использующего класс Stack

```
0 new #2 <generics_test/Main$Stack>
3 dup
4 invokespecial #3 <generics_test/Main$Stack.<init>>
7 astore_1
8 aload_1
9 ldc #4 <abc>
11 invokevirtual #5 <generics_test/Main$Stack.push>
14 return
```

java.lang.Comparable: из класса извлекается ссылка на виртуальный метод *compareTo,* который реализуется данным классом, и этот метод вызывается по ссылке командой *invokeInterface*. В терминах виртуальной машины: из объекта x извлекается его фактический тип, а из него, в свою очередь, – его *таблица виртуальных методов (Virtual Method Table – VMT)*, в которой для фактического типа и любого его потомка указатель на метод *compareTo* имеет фиксированное *смещение (индекс)*. Смещение виртуального метода в VMT его класса – это и есть информация о типе, по которой во время исполнения программы можно получить ссылку на фактическую характеристику типа – его метод. В данном случае фактическим типом является тип *String*.

Таким образом, благодаря явному представлению VMT во время исполнения, нет необходимости в явной передаче указателя на метод *compareTo* в качестве аргумента.

Проблема в данной схеме, как и вообще в реализации параметризованных типов в Java, лишь одна, но весьма серьезная – *стирание типа (type erasure)*: явного представления типа во время выполнения не существует. В качестве указателя типа любого объекта выступает ссылка на класс объекта (точнее говоря, на образ данного класса, загруженный в память виртуальной машины). К сожалению, для конкретного типа *Stack<String>* никакого явного представления во время исполнения программы не существует.

Остальные команды байт-кода метода *push* здесь подробно не рассматриваются, так как носят вспомогательный характер – реализуют вывод результата вызова метода *compareTo*.

5.6. Параметризация и подтипы. Согласно общему *принципу подстановки* (*Liskov substitution principle*), сформулированному Б. Лисков [13], любое свойство, которым обладает некоторый тип, должно выполняться для любого его *подтипа* (но не наоборот!). В объектно-ориентированном программировании (ООП) в качестве подтипов любого объектного типа выступают его прямые или косвенные *классы-потомки (подклассы)*. Основное свойство классов в ООП – наличие и доступность в них определенных *методов*: если нестатический метод M есть у класса C, то такой же метод есть и у подкласса (наследуется им). Другое важнейшее свойство – возможность *присваивания ссылок на объекты подкласса переменным типа класса (ссылок на потомков переменным типа-предка)*: если $C\ p$ – переменная типа-предка, а $D\ q$ – переменная типа-потомка, то возможно присваивание $p = q$, при котором, однако, для объекта через переменную p могут быть выполнены только методы, имеющиеся у типа C.

Листинг 9. Байт-код метода *push* параметризованного класса Stack

```
0 aload_1
1 aload_1
2 invokeinterface #2 <java/lang/Comparable.compareTo>
  count 2
7 istore_2
8 getstatic #3 <java/lang/System.out>
11 new #4 <java/lang/StringBuilder>
14 dup
15 invokespecial #5 <java/lang/StringBuilder.<init>>
18 ldc #6 <i=>
20 invokevirtual #7 <java/lang/StringBuilder.append>
23 iload_2
24 invokevirtual #8 <java/lang/StringBuilder.append>
27 invokevirtual #9 <java/lang/StringBuilder.toString>
30 invokevirtual #10 <java/io/PrintStream.println>
33 return
```

Как подчеркивают авторы концепции параметризованных типов в Java, операция параметризации *не* сохраняет данное отношение «тип-подтип». Например, если *G <T>* – определение *параметризованного типа*, *C* – некоторый *класс*, а *D* – *подкласс* класса *C*, то конкретный тип *G<D> не является* подтипом типа *G<C>*. Поэтому оба присваивания в коде, приведенном ниже:

```
p: G<C>; q: G<D>; p = q; q = p;
```

ошибочны (ошибка фиксируется во время компиляции).

Кроме того, в языке Java, если параметризованный класс *G1* – подкласс параметризованного класса *G:*

```
public class G1<T> extends G<T> { … }
```

то тип *G1<C> не является подтипом типа G<C>* (при любом *C*), так что присваивания в приведенном ниже коде:

```
r: G1<C>; s: G<C>;   r = s; s = r;
```

также *ошибочны*.

С практической точки зрения, такое ограничение вполне естественно. Например, если разрешить отношение тип-подтип в первом случае, то при присваивании типу-«предку» ссылки на тип-«потомок» ослабляются ограничения на объект, который разделяют переменные типа-предка и типа-потомка, поэтому через переменную с более слабыми ограничениями может быть выполнена над объектом недопустимая операция. Весьма выразительный пример приводит автор спецификации параметризованных типов в Java G. Bracha (Sun): если *Driver* – подтип типа *Person*, то, если допустить, что *List<Driver>* – подтип типа *List<Person>* (тип «список водителей» – подтип типа «список людей»), то с помощью операции добавления элемента в список над объектом-списком водителей, типизированным как *List<Person>*, в этот список несанкционированным образом могут быть добавлены люди, не являющиеся водителями (то есть не имеющие водительских прав). Данный пример относится, разумеется, только к теории типов и к спецификации типов в языке Java: в повседневной житейской практике, к сожалению, подобное происходит нередко, причем может иметь тяжелые последствия.

Для лучшего понимания объясненной концепции приведем несколько ошибочных вариаций на тему стека (см. пример 7):

```
Stack <String> s = new Stack <String>();
Stack <Integer> i = s;  // ошибка
Stack<Object> so = s;   // ошибка
s = so;                 // ошибка
MyStack<String> ms = s; // ошибка
```

Здесь предполагается, что *MyStack* – подкласс параметризованного класса *Stack*.

5.7. Шаблоны (wildcards). Ввиду описанных выше ограничений статической типизации, возникают некоторые проблемы полиморфизма методов, которые решаются введением в Java дополнительной конструкции – *шаблона (wildcard)*, означающего «любой тип». Шаблон обозначается символом «?» (вопросительный знак).

Пусть, например требуется написать метод *printCollection*, печатающий коллекцию элементов *любого* типа. Никакие прямолинейные приемы, например, указание в качестве типа аргумента типа *Collection <Object>*, не помогают: например, передача в данный метод аргумента типа *Collection <String>*, аналогичная по семантике присваиванию объектных ссылок, приводит к ошибке времени компиляции.

Следовательно, в данном случае требуется дополнительная возможность – *шаблоны*. С использованием шаблонов метод *printCollection* может быть реализован следующим образом:

```
void printCollection (Collection <?> c) {
   for (Object e:  c) {
     System.out.println(e);
}
```

По определению семантаки шаблонов, любой тип *Collection <T>* считается совместимым с типом *Collection <?>*.

Возможно также использование *шаблонов с ограничениями (bounded wildcards)*, в которых шаблон «?» дополняется ограничениями, проверяемыми для фактического типа:

```
<? extends C>
```

Здесь для фактического типа-параметра проверяется, что он является подклассом класса *C*.

В приведенном ниже примере, принадлежащем фирме Sun (пример 10), с помощью шаблонов с ограничениями определяется метод *drawAll,* позволяющий нарисовать любую геометрическую фигуру, представляемую каким-либо классом-потомком абстрактного класса *Shape – Circle, Rectangle, Canvas* (см. листинг 10).

5.8. Предложения по расширению параметризованных типов в Java. Проведенный в работе анализ концепции параметризованных типов в Java показывает, что необходимы ее расширения, которые позволили бы сделать ее более современной, удобной, надежной и безопасной. Данная статья – лишь начало той работы по спецификации и реализации предлагаемых расширений параметризованных типов в Java, которая может потребовать длительного времени (до нескольких лет). По существующим правилам расширения Java-технологии, действующим в *Java Community Process (JCP)* – некоммерческой организации, обсуждающей и утверждающей все расширения Java, необходимо:

– сформулировать спецификацию нового расширения Java-технологии – *Java Specification Request (JSR)*;

– разработать его эталонную реализацию (*Reference Implementation – RI*) в рамках открытого проекта *openJDK,* на основе которого в настоящее время корпорация Oracle выпускает все новые версии Java (первая такая версия – Java 7, выпущенная в ноябре 2011 г.);

– разработать комплекс тестов для его тестирования *(Technology Compatibility Kit – TCK)*;

– добиться принятия нового расширения большинством голосов заинтересованными членами JCP.

Необходима также организационная и финансовая поддержка от *спонсора*. Все эти вопросы еще предстоит решить.

Однако, несмотря на серьезные организационные аспекты, данный подход представляется реальным, ввиду того, что проф. В.О. Сафонов с 2011 года принят в члены JCP и широко известен как специалист в области Java-технологии, участвовавший в разработке JDK. Мы рассчитываем на поддержку, в первую очередь, корпорации Oracle.

В данном разделе сформулированы принципы предлагаемых расширений и описан синтаксис параметров-констант.

Мы считаем, что параметризованные типы в Java необходимо расширить следующим образом:

1. Минимальное расширение – параметры-константы. Опыт предыдущих разработок и использования языков программирования, начиная еще с 1960–1970-х гг., пока-

Листинг 10. Шаблоны с ограничениями и их использование

```
public abstract class Shape
  { public abstract void draw(Canvas c); }
public class Circle extends Shape
   { private int x, y, radius;
   public void draw(Canvas c) { ... }
}
public class Rectangle extends Shape
  { private int x, y, width, height;
  public void draw(Canvas c) { ... }
}
public class Canvas {
  public void draw(Shape s) { s.draw(this); }
}
public void drawAll(List<? extends Shape> shapes) { ... }
```

зывает, что параметры-константы, наряду с параметрами-типами, – весьма удобная возможность и полезная часть механизма параметризованных типов. Поэтому их отсутствие не только в Java, но и в .NET (включая C#), вызывает удивление. Наши вопросы к разработчикам .NET еще в 2002 г. на эту тему ни к чему не привели: разработчики считают, что в параметрах-константах нет необходимости, что, на наш взгляд, ошибочно. Синтаксис предлагаемых параметров-констант в параметризованных типах данных Java описан ниже.

2. Введение динамической информации о типах во время исполнения, включая и конкретизации параметризованных типов. «Стирания типов» допускать нельзя, так как оно противоречит принципам надежного и безопасного программирования (trustworthy computing) [14]. В соответствии с этими современными принципами, информация о типах данных должна быть доступна на любой фазе обработки и исполнения программ.

3. Вызов методов фактического параметра-типа, заданных в ограничениях на тип-параметр, с помощью новой виртуальной команды JVM invokedynamic. Данная команда введена в Java 7 для поддержки динамических языков с расширяемыми во время выполнения типами данных. По семантике использование команды ***invokedynamic*** вполне подходит для реализации таких вызовов, однако требуется более полный и детальный анализ.

4. Формальная спецификация параметризованных типов по принципам контрактного проектирования (design-by-contract) на основе предусловий, постусловий и инвариантов. Аналогичные возможности реализованы для платформы .NET в системе Spec# [4], разработанной Microsoft Research, – расширении языка C# средствами формальных спецификаций. Такое расширение позволит осуществить, наконец, для Java-технологии принцип *доказательного программирования*, к воплощению которого, как видно из предыдущих разделов данной работы, стремились классики в области ИТ, начиная с 1970-х гг.

Предлагаемый синтаксис параметров-констант в языке Java:

```
public class Stack
 <type t, const int maxDepth>
where t implements SomeInterface &&
              maxDepth > 0 &&
              maxDepth <= 1000
  … if (sp >= maxDepth) { throw new Full(); ) …

Stack<String, 5000> s = …
// фактический параметр -
// константное выражение
```

Данный код определяет параметризованный тип (класс) *Stack* с *двумя* параметрами – *типом элемента* стека *t* и *максимальной глубиной* – константой *maxDepth* (целым числом). Соответственно, расширены и ограничения на параметры класса *Stack* – кроме ограничений на параметр-тип, заданы также *ограничения на параметр-константу maxDepth*: она должна быть положительна и не должна быть больше 5000.

Предлагаемая реализация параметров-констант. Параметры-константы вполне естественно вписываются в схему реализации Java и организации выполнения Java-программ и не усложняют ее. Фактический параметр-константа (например, значение 100) может быть передана как *дополнительный неявный аргумент* любого метода параметризованного класса. Например, для метода *push* класса *Stack* с двумя параметрами – типом элемента и максимальной глубиной стека – состав фактических аргументов будет следующим:

```
push:  arg0 = this, arg1 = maxDepth, arg2 = x
```

ЗАКЛЮЧЕНИЕ

Мы считаем концепцию параметризованных типов и ее предлагаемое расширение в языке Java (параметры-константы, динамическая информация о типах, формальные спецификации) перспективным направлением на пути к разработке и многократному использованию верифицированных реализаций алгоритмов в форме пакетов, интерфейсов, классов и (в будущих версиях Java) модулей.

Авторы будут благодарны за замечания и предложения по статье и открыты к предложениям о поддержке описанных в статье идей от корпорации Oracle и других компаний – членов JCP и авторов новых концепций в области Java-технологии.

Литература

1. *Хоар Ч.* Доказательство корректности представления данных // В сб.: Данные в языках программирования. Под ред. Агафонова В.Н. М.: Мир, 1982.
2. *Лисков Б., Гаттэг Дж.* Использование абстракций и спецификаций при разработке программ. М.: Мир, 1989.
3. *Сафонов В.О.* Языки и методы программирования в системе «Эльбрус». М.: Наука, 1989.
4. *Safonov V.O.* Trustworthy Compilers. – Wiley International. John Wiley & Sons, 2010.
5. *Дал У., Мюрхауг К., Нюгорд К.* Универсальный язык программирования Симула-67. М.: Мир, 1968.
6. Modals (mode alternatives): Proposal by C.H. Lindsey (74-04-05), Algol Bulletin, 37 . 4 . 3, CAM – 6, Partial Parametrization and Modals
7. *Shaw M., Wulf W., London R.* Abstraction and verification in ALPHARD: Defining and specifying iteration and generators. – Communications of the ACM. Vol. 20, Issue 8, August 1977. P. 553–564.
8. *Floyd R.W.* Assigning Meaning to Programs, in Proceedings of Symposium on Applied Mathematics, Vol. 19, J.T. Schwartz (Ed.), A.M.S., 1967. P. 19–32.
9. Web-сайт системы Spec# / http://research.microsoft/com/specsharp (дата обращения: 22.01.2012).
10. *Бар Р.* Язык Ада в проектировании систем. М.: Мир, 1982.
11. Web-сайт языка Visual Prolog / http://www.visual-prolog.com (дата обращения: 22.01.2012).
12. *Сафонов В.О.* Введение в Java-технологию. Lambert Academic Publishing, Saarbrucken, Germany, 2011.
13. *Liskov B.* Data abstraction and hierarchy. SIGPLAN Notices, May 1988.
14. *Safonov V.O.* Using aspect-oriented programming for trustworthy software development. Wiley International. John Wiley & Sons, 2008.

Abstract

The article covers the concept of generic data types that plays key role in modern programming, and the proposal by the authors on extending generics in Java. The article is based on the talk at the International Conference EclipseCon 2011 Europe, Ludwigsburg, Germany, November 2011.

Keywords: parametrized data types (generics), abstract data types, templates, modals, generic packages, Simula 67, Ada, Java, C++, C#.

Сафонов Владимир Олегович,
доктор технических наук, профессор кафедры информатики математико-механического факультета СПбГУ,
vosafonov@gmail.com,

Сафонова Адель Наркисовна,
младший научный сотрудник лаборатории Java-технологии математико-механического факультета СПбГУ, программист,
adel_safonova@mail.ru