МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра "ПОВТ и АС"

Обобщения (Generics) в Java

Методические указания к лабораторной работе

по дисциплинам "Программирование", "Объектно-ориентированное программирование"

Ростов-на-Дону 20 г.

Составитель: к.ф.-м.н., доц. Габрельян Б.В.

УДК 512.3

Обобщения (Generics) в Java: методические указания – Ростов н/Д: Издательский центр ДГТУ, 20 . – 8 с.

В методической разработке рассматриваются общий синтаксис описания и использования обобщенных классов и методов, стандартные коллекции Java. Даны задания по выполнению лабораторной работы. Методические указания предназначены для студентов направлений 090304 "Программная инженерия", 020303 "Математическое обеспечение и администрирование информационных систем".

Ответственный редактор:

Издательский центр ДГТУ, 20

Обобщения в Java это средство поддержки параметрического полиморфизма. Параметрический полиморфизм позволяет отвлечься от конкретных типов при разработке классов, интерфейсов или методов и обобщить их на широкий круг типов. Например, общие операции для списка целых значений или списка объектов какого-либо класса A одинаковые: добавить новый элемент в список, удалить указанный элемент из списка, получить заданный элемент списка, узнать пуст ли сейчас список. Нужно ли нам тогда явно создавать два класса "Список", один для целых значений, другой для объектов A? Заменив конкретный тип (целый или A) параметром можно дать одну реализацию структуры данных "Список", переложив на компилятор заботу по правильной работе со списками элементов разных типов. Это и позволяют сделать обобщения.

1. Синтаксис описания обобщенных классов.

При учете товаров (услуг и т.п.) используется различная информация, название товара, его количество и, кроме того, идентификационный номер. Например,

```
class Product1 {
        String name;
        int quantity;
        int id;
        …
        public int getId() { return id; }
};
```

или

```
class Product2 {
        String name;
```

```
        int quantity;
        String id;
        …
        public String getId() { return id; }
    };
```

Разница в реализациях классов Product1 и Product2 лишь в типе идентификатора id. А если понадобится идентификатор другого типа (не int и не String) придется создавать еще один класс. Нельзя ли решить задачу с помощью всего одной реализации (одного класса)? Идея может заключаться в использовании в качестве идентификатора ссылки на тип Object, тогда ее можно будет связывать с объектом любого класса (типа). Правда, при работе с id нужно будет всякий раз преобразовывать ссылку на Object к ссылке на конкретный тип (Integer, String, …).

```
    class Product {
        String name;
        int quantity;
        Object id;
        …
        public Object getId() { return id; }
    };
…
Product p = new Product();
int id = (Integer)p.getId(); // явное приведение типа
```

Здесь неудобной и потенциально небезопасной является необходимость явного приведения типа. С одной стороны, случайно или преднамеренно, ссылка на id может быть связана с объектом неподходящего типа, с другой, может быть выполнена попытка привести ссылку id к неправильному типу. Обобщенные типы как раз и избавляют программиста от необходимости явно выполнять контроль и правильное приведение типов. Этим будет заниматься компилятор.

Обобщенный класс можно задать следующим образом

```
class Product<T> {
        String name;
        T quantity;
        T id;
        …
        public T getId() { return id; }
};
```

Это описание класса. При его использовании параметр T должен быть заменен конкретным типом.

```
…
Product<Integer> p1 = new Product<Integer>();
int id1 = p1.getId(); // не нужно делать явное приведение типа
```

начиная с Java 7 можно использовать более короткую запись

```
Product<Integer> p1 = new Product<>();
int id1 = p1.getId();
Product<String> p2 = new Product<>();
```

String id2 = p2.getId();

Типы Product<Integer> и Product<String> разные, это видно на этапе компиляции, поэтому, например,

~~**p1 = p2;**~~

вызовет сообщение о синтаксической ошибке.

В Java обобщенные типы нельзя строить на основе простых встроенных типов (типов-значений), таких как int, float, boolean. Можно использовать только ссылочные типы. В обобщенном типе параметр T с одной стороны, может замещаться любым типом, но, с другой стороны, у компилятора нет информации о том, какие методы (поля и т.д.) есть у типа, который будет замещать этот параметр. Возможно только копирование и присваивание ссылок. Поэтому в реализации обобщенного типа мы не можем вызывать какие-то методы, если не наложить на параметр ограничений.

Например, нужен обобщенный класс, представляющий понятие полином n-й степени. Тип полинома может быть, например, действительным или комплексным. Понятно, что это типы, представляющие числа. Мы знаем, что все классы-оболочки для стандартных числовых типов являются наследниками класса Number. Тогда ограничение параметра обобщенного типа может выглядеть следующим образом

```
class Polynomial<T extends Number> {
        private T[] a; // массив коэффициентов полинома
        …
}
```

Теперь можно создавать полиномы с элементами, типы которых классы, производные от Number.

Polynomial<**Integer**> poly1 = new Polynomial<>();

Polynomial<**Double**> poly2 = new Polynomial<>();

Но попытка создать полином для строк, завершится сообщением о синтаксической ошибке

~~Polynomial<**String**> poly3 = new Polynomial<>();~~

В Java типы, замещающие параметр видны на стадии компиляции, но после компиляции, на стадии выполнения программы будет существовать только один класс Polynomial. В реализации этого класса все ссылки на T будут заменены на ссылки на Object. Произойдет так называемое стирание типа (type erasure). Это означает в частности, что в методах обобщенного класса, например, Polynomial нельзя создавать объекты или массивы объектов типа T.

```
class Polynomial<T extends Number> {
        private T[] a; // массив коэффициентов полинома
        …
        public Polynomial(int n) {
                a = new T[n+1];
        }
}
```

Ведь динамическое выделение памяти с помощью операции new выполняется не на этапе компиляции, но во время выполнения программы, а на этом этапе все ссылки - это ссылки на Object, а не на конкретный тип. Таким образом объекты реальных типов должны создаваться вне методов обобщенных

классов, а самим этим методам должны передаваться ссылки на эти объекты. В случае массива, мы можем в методе обобщенного класса создать массив ссылок на Object и потом заносить в этот массив ссылки на объекты конкретных классов, пользуясь тем, что все эти классы неизбежно потомки Object.

```
class Polynomial<T extends Number> {
    private T[] a; // массив коэффициентов полинома
    …
    public Polynomial(int n) {
        a = (T[])new Object[n+1]; /* обязательное явное приведение от типа Object[] к типу T[] */
    }
    public void set(T value, int index) { /* value - ссылка на объект созданный вне Polynomial */
        … // проверка на допустимое значение index
        a[index] = value;
}
…
// y(x) = 2*x²+4*x+1
Polynomial<Integer> poly1 = new Polynomial<>(2);
poly1.set(1, 0);
poly1.set(4, 1);
poly1.set(2, 2);
```

Для обобщенного класса можно задавать несколько параметров

class Pair<**K, V**> { … }

Параметр обобщенного типа - это обычный идентификатор.

Обобщенный класс может быть классом-наследником необобщенного класса

```
class A {
        private int a;
        public A(int x) { a = x; }
        public int getA() { return a; }
}
class B<T> extends A {
        public B(int x) { // так как в A нет конструктора без параметров
                super(x); // необходимо вызвать конструктор базового класса
                        // этот вызов должен быть первой командой в конструкторе
        }
}
```

Необобщенный класс может быть наследником обобщенного, если для параметра последнего задан конкретный тип

```
class SuperPolynomial extends Polynomial<Double> {
        …
}
```

2. Обобщенные методы

Помимо классов, можно объявлять также обобщенные методы. Такие методы могут объявляться как в обобщенных, так и в необобщенных классах.

Например, имеются три класса

```
class A {
	public int a = 1;
	public int getA() { return a; }
}
class A1 extends A {
	public int a = 11;
	public int getA() { return a; }
}
class A2 extends A {
	public int a = 12;
	public int getA() { return a; }
}
```

И обобщенный класс

```
class B<T extends A> {
	public int a = 2;
	public int add(T da) {
		a += da;
		return a;
	}
}
…
```

B<A1> b = new B<>();

System.out.println(**b.add(new A1()**); // выводит 13

Но вызов

~~**b.add(new A2());**~~

вызывает сообщение о синтаксической ошибке. Причина понятна. При создании объекта b параметр T всюду в обобщенном классе B приводится к типу A1, в том числе и в методе add, поэтому add не может принять аргумент типа A2. Но по своему функционалу классы A1 и A2 идентичны (они получили его от родительского класса A). Чтобы разрешить методу add использовать объекты любых классов, производных от класса A, нужно его обобщить (параметризовать).

```
class B<T extends A> {
    public int a = 2;
    public <E extends A> int add(E da) { /* имя параметра метода E отличается от имени параметра класса T */
        a += da;
        return a;
    }
}
…
B<A1> b = new B<>();
System.out.println( b.add(new A2() ); // выводит 14
```

4. Обобщенные интерфейсы

Интерфейсы также могут быть обобщенными. Синтаксис такой же, как и для обобщенных классов.

```
interface Func<T, R> {
    R call(T arg);
}
class MyFunc implements Func<Double, Double> {
    public Double call(Double x) { return Math.sin(x); }
};
```

5. Стандартные коллекции Java

Все классы и интерфейсы коллекций в Java, начиная с версии 5 обобщенные. Например, класс ArrayList<T> представляет понятие "динамический массив".

У этого класса есть следующие конструкторы:

ArrayList()

ArrayList(int initialCapacity); /* количество элементов, под которые зарезервирована память */

Изначально массив пуст.

Основные методы:

int size() текущий размер

boolean isEmpty() проверка на пустоту

boolean contains(Object o) содержит ли объект o

T get(int index) возвращает элемент в позиции index, индексирование с нуля

T set(int index, T elem) /* заменяет элемент в позиции index на elem, возвращает старое значение */

int indexOf(Object o) индекс первого вхождения объекта o

void add(T elem) добавить в конец массива

void add(int index, T elem) добавить в позицию index

void remove(int indtx) /* удаляет элемент в позиции index и возвращает ссылку на него */

boolean remove(Object o) удаляет объект o

void clear() удаляет все элементы

Помимо обобщенного, существует и необобщенная версия ArrayList. Он существовала и в версиях Java до версии 5. Этот класс хранит ссылки на Object. Будет использоваться если не задать угловые скобки после имени класса.

ЗАДАНИЕ 1. Создайте обобщенный класс Pair, представляющий понятие "пара значений". Такой класс оказывается полезным в ситуациях, когда нужно использовать два связанных друг с другом значения. Например, алгоритм поиска наибольшего элемента заданного массива может вернуть само это наибольшее значение и значение позиции (индекса) первого элемента массива с таким значением. Типы элементов пары могут быть произвольными (но ссылочными) и в общем случае не совпадающими. Необходимо предусмотреть конструктор (или конструкторы) для инициализации вновь созданных пар, возможность получение/изменения каждого из

значений пары (можно, как это сделано, например, в стандартном шаблоне pair в C++ просто использовать открытые поля first и second) и метод make_pair для создания пары значений, который можно вызывать даже если у нас пока нет ни одного объекта класса Pair.

ЗАДАНИЕ 2. Мешок (Bag) - это емкость фиксированного размера, в которую можно складывать различные предметы. Поднимая мешок мы его встряхиваем и предметы перераспределяются в мешке произвольным образом. Из мешка достается тот предмет, который подвернулся под руку первым, то есть какой-то из имеющихся, но неизвестно какой. Другими словами, Bag - это контейнер, в который новые элементы добавляются в произвольную (будем считать случайную) позицию и удаляются из случайно выбранной позиции. Создайте (НЕ ОБОБЩЕННЫЙ!) класс Bag в котором элементы хранятся в массиве (какого типа должны быть элементы массива если по требованиям задачи Bag может хранить предметы любых типов?). Размер массива (предельный размер конкретного "мешка") указывается в конструкторе при создании объекта и дальше меняться не может. Нужно предусмотреть методы для добавления, удаления (этот метод должен возвращать удаленный элемент) элементов, метод возвращающий (какой-то) элемент и метод, возвращающий значение текущего размера Bag. Какие методы (или один метод) Bag не должны переопределяться в классах-потомках? Запретите их (или его) переопределение.
Метод random() класса Math возвращает псевдослучайное действительное значение в диапазоне от 0.0 до 1.0, включая 0.0 но не включая 1.0. Чтобы получить псевдослучайное целое значение из диапазона [0, size-1] можно использовать выражение (int)Math.round(Math.random()*(size-1)).
Протестируйте работоспособность класса Bag для элементов разного типа (Integer, String, …).

ЗАДАНИЕ 3. Создайте НЕОБОБЩЕННЫЙ класс PairBag, представляющий "мешок" для хранения пар значений. Используйте созданные раньше обобщенный класс Pair и необобщенный класс Bag. Прежде всего, определите, какое отношение между классами Bag и PairBag следует использовать. Как и следует из названия, объекты PairBag должны хранить только пары значений (не могут хранить отдельные) значения. Методы должны работать с парами (получать/возвращать).

При этом, в одном и том же объекте PairBag одновременно могут храниться могут пары с разными типами значений, например, Pair<Integer,Integer> и Pair<Integer,String>.

ЗАДАНИЕ 4. Хранение в контейнере пар разного типа обычно приводит к неопределенности при использовании пар. Такой проблемы не будет, если типы пар одинаковые. Если тип первого значения пары T1, а тип второго значения пары T2, то все пары в контейнере должны иметь тип Pair<T1,T2>. Создайте обобщенный класс GPairBag для хранения пар одинакового типа. В реализации не используйте стандартные контейнеры Java. Используйте те классы, которые Вы реализовали при решении предыдущих заданий этой лабораторной работы. Начните с определения того, какое отношение между GPairBag и PairBag удобно использовать.

ЗАДАНИЕ 5. Создайте обобщенный класс GenericPairBag для хранения пар одинакового типа, используя подходящий стандартный обобщенный контейнерный класс Java, например, ArrayList.

ЗАДАНИЕ 6. Проводится спортивное соревнование, турнир, среди N команд. Нужно провести жеребьевку, то есть определить пары команд, которые будут проводить игры друг с другом. Для этого записки с названиями команд складываются в "мешок", перемешиваются, затем последовательно извлекаются из мешка и две подряд выбранные записки определяют пару играющих друг с другом команд. Получившиеся пары помещаются в другой "мешок". Затем, пока в этом "мешке" еще что-то есть, из него выбирается какая-то пара и пользователю задается вопрос о том, какая команда выиграла, первая или вторая (игра проводится до победы одной из команд, ничьей быть не может). Названия выигравших команд складываются в первый "мешок". Теперь их будет N/2 штук. Процесс продолжается до тех пор, пока не останутся две команды. Победившая команда является победителем всего турнира. Напишите программу, реализующую описанный выше процесс проведения турнира. Число команд-участников N задается пользователем. Очевидно, что N не может быть произвольным. Если пользователь дает недопустимое значение N можно задать значение по умолчанию, например, 8. "Мешки" должны быть представлены каким-нибудь подходящим стандартным классом-контейнером (или классом Bag) и классом GenericPairBag. Имена

команд можно генерировать автоматически ("Команда1", "Команда2", …).

ЗАДАНИЕ 7. Создайте класс DList, содержащий два поля - список значений произвольного типа T1 и список списков значений произвольного типа T2. Работа с полями должна быть согласованной, то есть, например, при добавлении новых значений должны указываться одновременно целое значение и список целых значений, соответственно целое значение добавляется в первое поле класса, список во второе, Таким образом, каждому элементу в позиции i первого поля (списка) соответствует список в позиции i второго поля (списка списков). Удалять и получать информацию можно указав либо позицию i, либо значение первого поля. Например, первое поле может содержать список {1, 2, 3, 4, 5}, с каждым элементом этого списка могут быть связаны, например, такие списки: {1}, {1,1}, {1,2}, { {1,3}, {2,2} }, {2,3}. То есть со всеми элементами, кроме 4, связано по одному списку, а со значением 4 связаны два списка.

ЗАДАНИЕ 8. Реализуйте алгоритм выдачи указанной суммы денег (целое значение) минимальным количеством монет из заданного набора методом динамического программирования. Алгоритм должен возвращать не только количество монет, но и все возможные комбинации монет, обеспечивающие оптимальную выдачу суммы. Используйте в реализации класс DList. Протестируйте работу алгоритма для разных наборов монет (например, 1, 2, 5, 10 и 1, 4, 7, 9). Здесь первое поле DList будет содержать набор промежуточных результатов, а второе поле - списки номиналов монет, которыми должна быть выдана сумма, соответствующая промежуточному результату. Пример в задании 7 соответствует решению задачи выдачи суммы 5 монетами номиналов 1, 2, 3.

Хэш-таблицы. Есть набор быть может сложно организованных данных. Нужно многократно искать информацию в этом наборе. В каком случае поиск будет максимально быстрым? Мы знаем, что очень быстро можно найти элемент в массиве, если известен индекс этого элемента. В этом случае сложность операции поиска не зависит от размера массива. Пусть, например, хранимые данные - это данные о конкретных людях: фамилия, возраст, телефоны и т.д. Поиск проводим по фамилии. Проблема в том, что существует очень большое количество разных фамилий, несоизмеримо больше потенциально возможных фамилий. Поставить в соответствие любой возможной фамилии элемент

массива невозможно, его размер окажется огромным. Но если мы можем отобразить каждый элемент некоторого большого набора (все возможные фамилии) в значение из какого-то ограниченного набора (набор индексов элементов массива), то есть у нас есть некоторая функция, выполняющая такое отображение (хэш-функция), то задачу быстрого поиска можно решить, построив такой массив. Очевидно, что отобразить элементы большого набора в меньший однозначно в общем случае невозможно. Неизбежно несколько элементов большого набора будут отображаться в одно и то же значение из малого набора. Значит для таких случаев для одного индекса массива нужно будет хранить несколько значений. Другими словами, можно организовать данные в виде хэш-таблицы, например, как массив списков. Информация хранится в элементах списков, а массив нужен для быстрого доступа к конкретному списку. Если хэш-функция отображает некоторую фамилию на индекс массива однозначно, соответствующий список состоит из одного элемента и весь поиск сводится к нахождению этого элемента. В противном случае в списке будут храниться все записи, отображенные на данный индекс, и чтобы найти нужные данные придется искать нужную запись в списке. Например, есть фамилия "ИВАНОВ". Хэш-функция (крайне неоптимальная) может просто сложить коды всех символов соответствующей строки и выдать целое число 6279 - индекс элемента массива. Для другой фамилии, например, "ЛОБАНОВ" та же хэш-функция вернет число 7335. Индексы разные - поиск быстрый. Но для фамилии "БАЛОНОВ", очевидно, значение хэш-функции такое же как для фамилии "ЛОБАНОВ" - 7335. Таким образом элемент таблицы с индексом 7335 будет содержать список из двух элементов и, чтобы найти информацию о человеке с фамилией "БАЛОНОВ" нужно будет еще и сравнивать фамилии для всех записей в этом списке.

ЗАДАНИЕ 9. Создайте обобщенный класс HashFunction<K> с одним объявленным абстрактным методом int hash(K s). При создании объекта класса HashFunction ему должна передаваться информация о размере таблицы – это определяет, какие целые значения (из какого диапазона) должна выдавать функция. Создайте обобщенный класс HashTable для представления хэш-таблицы в виде массива списков. Учтите, что в таблице хранятся данные одного типа T, а поиск в таблице может осуществляться по другому типу K (например, поиск проводится не по всей записи, а только по одному ее полю). Это, в частности накладывает ограничение на тип элементов таблицы. Нужно уметь проверять, что конкретный элемент содержит значение (например, отдельное поле) по которому хэш-функция генерирует свое значение. То есть тип T должен содержать реализацию некоторого метода (например, boolean

contains(K value)), объявленного в определенном базовом классе (или в реализуемом интерфейсе). Размер массива (таблицы) и конкретная хэш-функция (как ссылка на объект класса-наследника HashFunction) передаются конструктору класса при создании объекта HashTable.

ЗАДАНИЕ 10. Создайте класс Person, для хранения информации о человеке: фамилия, возраст и другие поля (какие и сколько решайте сами). Создайте класс-наследник HashFunction с какой-нибудь реализацией хэш-функции, получающей фамилию человека и возвращающей целое значение (можно использовать алгоритм, описанный выше немного модифицировав его так, чтобы учесть максимально возможный размер массива, задаваемый извне). Напишите программу для тестирования работы хэш-таблицы, содержащей объекты класса Person.

Литература

1. Дж.Гослинг, Б.Джой, Г.Стил, Г.Брача, А.Бакли "Язык программирования Java SE 8. Подробное описание, 5-е изд.". – М.: Вильямс. – 2016, 672с.
2. Н.А.Прохоренок "Основы Java, 2-е изд.". – СПб.:BHV-Петербург. – 2019, 768с.
3. Б.Эккель "Философия Java. Библиотека программиста". – СПб.: Питер. – 2001, 880с.
4. П.Ноутон, Г.Шилдт "Java 2". – СПб.:BHV-Петербург. – 2001, 1072с.
5. Д.Бишоп "Эффективная работа: Java 2". – СПб.:Питер; К.:BHV. – 2002, 592с.

Редактор А.А. Литвинова

ЛР № 04779 от 18.05.01. В набор В печать

Объем 0,5 усл.п.л., уч.-изд.л. Офсет. Формат 60x84/16.

Бумага тип №3. Заказ № Тираж 120. Цена

Издательский центр ДГТУ

Адрес университета и полиграфического предприятия:

344010, г. Ростов-на-Дону, пл. Гагарина, 1.