Лабораторная работа №5
«Монады в языке Java»

Скоробогатов С.Ю.

17 марта 2016 г.

1 Цель работы

Приобретение навыков использования монад Optional и Stream в программах на языке Java.

2 Исходные данные

Понятие *монады*, пришедшее из теории категорий, широко используется в функциональном программировании. Мы не будем углубляться в математические основы этого понятия, но рассмотрим два обобщённых класса – Optional и Stream, появление которых в стандартной библиотеке Java версии 8 оказало существенное влияние на стиль программирования на этом языке.

2.1 Монада Optional

Как показывает опыт, огромную проблему при программировании на Java создаёт наличие в языке нулевой объектной ссылки – **null**.

Исторически так сложилось, что нулевая ссылка широко используется в программах на Java как индикатор отсутствия значения. При этом предполагается, что программист должен помнить, что некоторая переменная может содержать нулевую ссылку, и вставлять в код программы соответствующие проверки при необходимости обращения к объекту, на который указывает эта переменная.

Поясним данный тезис на примере метода get контейнерного класса HashMap<K, V>:

```
public V get(Object key)
```

Этот метод возвращает ссылку на объект, в который хеш-таблица отображает ключ key. Причём в случае, если словарная пара с ключом key отсутствует в хеш-таблице, метод get возвращает **null**.

По идее, при обращении к объекту, ссылку на который вернул метод get, необходимо проверять, не является ли эта ссылка нулевой, но об этом часто забывают. Это не вызывает большой беды, если попытка обратиться к объекту происходит сразу после получения нулевой ссылки, так как мы получим в этом месте программы её аварийное завершение по NullPointerException и быстро обнаружим ошибку. Однако, часто бывает, что нулевая ссылка передаётся куда-то дальше, и обращение к ней происходит совсем в другом месте программы, никак на первый взгляд не связанным с пресловутым вызовом метода get хеш-таблицы.

2.1.1 «Заворачивание» значения в Optional<T>

Чтобы не использовать нулевые ссылки для индикации отсутствия значения, можно применить появившийся в стандартной библиотеке Java 8 обобщённый класс Optional<T>. Объект этого класса является контейнером для нуля или одного объекта класса T. Подразумевается, что объект Optional<T> либо является пустым, либо содержит ненулевую ссылку на объект класса T.

Создание объекта Optional<T> осуществляется с помощью одного из трёх статических методов:

1. `static <T> Optional<T> empty()`

   Возвращает пустой объект Optional<T>.

2. `static <T> Optional<T> of(T value)`

   «Заворачивает» ненулевую объектную ссылку value в объект Optional<T>.

3. `static <T> Optional<T> ofNullable(T value)`

   Если объектная ссылка value ненулевая, «заворачивает» её в объект Optional<T>. В противном случае конструирует пустой объект Optional<T>.

По идее, начиная с Java версии 8 метод get класса HashMap<K, V> должен был бы возвращать Optional<V>. В целях обратной совместимости разработчики Java не стали менять метод get, поэтому его приходится использовать в сочетании с методом Optional.ofNullable. Например:

```
HashTable<String, Integer> tab = new HashTable<>();
...
Optional<Integer> i = Optional.ofNullable(tab.get("qwerty"));
```

2.1.2 «Разворачивание» Optional<T>

«Развернуть» объект Optional<T> можно с помощью метода get. Этот метод возвращает ненулевую объектную ссылку, содержащуюся в непустом объекте Optional<T>. В случае пустого объекта порождается исключение NoSuchElementException.

Естественно, перед вызовом метода get следует убедиться, что объект Optional<T> – непустой, с помощью метода isPresent. Например:

```
Optional<Integer> i = Optional.ofNullable(tab.get("qwerty"));
if (i.isPresent()) System.out.println(i.get());
```

Нетрудно догадаться, что такой сценарий работы с Optional<T> мало чем отличается от использования нулевых ссылок. Действительно, мы могли бы обойтись без Optional<T>, переписав вышеприведённый фрагмент кода как

```
Integer i = tab.get("qwerty");
if (i != null) System.out.println(i);
```

Оба варианта имеют один и тот же недостаток: при написании кода нужно не забыть проверить, существует значение или нет. При этом вариант с Optional<Integer> всё же немного лучше – сам тип переменной i как бы намекает, что значения может и не быть.

К счастью, «разворачивание» Optional<T> путём вызова метода get необязательно, потому что класс Optional<T> имеет набор методов, позволяющих абстрагироваться от возможного отсутствия значения. В частности, выполнить какое-нибудь действие со значением, «завёрнутым» в Optional<T>, можно с помощью метода

```
void ifPresent(Consumer<? super T> consumer)
```

Этот метод принимает в качестве параметра лямбда-выражение, кодирующее действие. Если значение существует в объекте Optional<T>, для него будет вызвано лямбда-выражение. Если объект Optional<T> – пустой, ничего не произойдёт.

Тем самым, мы можем переписать вышеприведённый пример значительно более элегантным способом:

```
Optional<Integer> i = Optional.ofNullable(tab.get("qwerty"));
i.ifPresent(v -> System.out.println(v));
```

Отметим, что лямбда-выражение вида v -> obj.method(v), которое всего лишь вызывает существующий метод некоторого объекта, может быть записано более кратко в виде так называемой *ссылки на метод*, имеющей вид obj::method. Принимая также во внимание, что в последнем примере переменная i не особенно нужна, мы можем оформить этот пример ещё более каноничным для Java 8 способом:

```
Optional.ofNullable(tab.get("qwerty"))
    .ifPresent(System.out::println);
```

2.1.3 Композиция частичных функций

Пусть необходимо выполнить композицию частичных функций $f(g(h(x)))$. Здесь каждая функция – это вызов некоторого метода.

Так как функции частичные, то они не определены для некоторых значений своих аргументов, т.е. могут возвращать нулевую ссылку. Но при этом каждая функция подразумевает, что передаваемое ей значение существует, т.е. выражается ненулевой ссылкой.

В качестве примера рассмотрим следующую ситуацию. Имеется три хеш-таблицы. Нужно заглянуть в первую таблицу по ключу x, взять оттуда значение y, использовать это значение как ключ для второй таблицы и взять соответсвующее ему значение z, а затем залезть в третью таблицу по ключу z:

```
public static String compose(
        HashMap<String, Integer> a,
        HashMap<Integer, Float> b,
        HashMap<Float, String> c,
        String x)
{
    if (x != null) {
        Integer y = a.get(x);
        if (y != null) {
            Float z = b.get(y);
            if (z != null) {
                return c.get(z);
            }
        }
    }
    return null;
}
```

Как видим, нам потребовался каскад **if**'ов из-за того, что любой из трёх вызовов метода get может вернуть **null**.

Метод map класса Optional<T> позволяет обойтись без проверок при записи композиции частичных функций:

```
<U> Optional<U> map(Function<? super T, ? extends U> mapper)
```

Метод map принимает лямбда-выражение, определяющее частичную функцию. Если объект Optional<T> пустой, то метод map ничего не делает и просто возвращает пустой Optional<U>. В противном случае значение, записанное в объекте Optional<T>, передаётся в лямбда-выражение, и метод map возвращает Optional<U>, в который «завёрнут» результат вычисления лямбда-выражения. При этом результат «заворачивается» по принципу работы метода ofNullable, т.е. если он представляет собой нулевую ссылку, то Optional<U> будет пустым.

Код приведённого выше метода compose можно переписать с использованием метода map. При этом будет логично изменить тип возвращаемого методом compose значения на Optional<String>:

```
public static Optional<String> compose(
        HashMap<String, Integer> a,
        HashMap<Integer, Float> b,
        HashMap<Float, String> c,
        String x)
{
    return Optional.ofNullable(x)
                .map(a::get).map(b::get).map(c::get);
}
```

2.2 Монада Stream

Интерфейс Stream<T> в некотором смысле является развитием идеи итераторов, т.е. представляет последовательность некоторых значений, называемую *потоком*.

Получить объект Stream<T> можно из объекта любого контейтерного класса стандартной библиотеки языка Java путём вызова метода stream:

```
Stream<T> stream()
```

Вытащить все значения из потока Stream<T> можно с помощью метода forEach:

```
void forEach(Consumer<? super T> action)
```

Этот метод получает в качестве параметра лямбда-выражение и вызывает его для каждого значения из последовательности, представляемой потоком.

В качестве примера составим обобщённый метод printList, предназначенный для вывода элементов объекта ArrayList в стандартный поток вывода:

```
public static <T> void printList(ArrayList<T> list) {
    list.stream().forEach(System.out::println);
}
```

2.2.1 Преобразование последовательностей

Основной смысл использования объектов Stream<T> для представления последовательностей заключается в том, что методы интерфейса Stream<T> позволяют преобразовывать последовательности.

Рассмотрим основные преобразования, определённые в интерфейсе Stream<T>:

1. `Stream<T> filter(Predicate<? super T> predicate)`

   Возвращает новый поток, содержащий только те элементы исходного потока, которые удовлетворяют переданному в качестве параметра предикату.

2. <R> Stream<R> map(Function<? **super** T, ? **extends** R> mapper)

   Получает в качестве параметра лямбда-выражение, принимающее значение типа T и возвращающее значение типа R. Возвращает поток Stream<R>, представляющий последовательность значений, получаемую путём вызова этого лямбда-выражения для каждого элемента исходного потока.

3. <R> Stream<R> flatMap
   (Function<? **super** T, ? **extends** Stream<? **extends** R>> mapper)

   Получает в качестве параметра лямбда-выражение, принимающее значение типа T и возвращающее последовательность значений типа R, представленную потоком Stream<R>. Возвращает конкатенацию потоков, получаемых путём вызова этого лямбда-выражения для каждого элемента исходного потока.

В качестве примера использования преобразований filter и map приведём решение следующей задачи. Пусть дан динамический массив ArrayList<Integer>. Требуется вывести в стандартный поток вывода последовательность чисел, получаемую путём возведения двойки в степени, заданные числами из этого динамического массива. При этом отрицательные степени должны пропускаться:

```
public static void powers(ArrayList<Integer> list) {
    list.stream()
            .filter(x -> x >= 0)
            .map(x -> 1 << x)
            .forEach(System.out::println);
}
```

Чтобы продемонстрировать применение преобразования flatMap, возьмём целочисленную матрицу ArrayList<ArrayList<Integer>> и распечатаем последовательность её ненулевых элементов:

```
public static void nonZeroes(ArrayList<ArrayList<Integer>> m) {
    m.stream()
            .flatMap(row -> row.stream().filter(x -> x != 0))
            .forEach(System.out::println);
}
```

2.2.2 Собирание последовательностей

Потребление значений из потоков выполняется так называемыми *коллекторами* – объектами классов, реализующими интерфейс Collector. Коллекторы предназначены для выполнения анализа потоков и записи результатов реализуемых потоками преобразований в объекты контейнерных классов.

Чтобы задействовать некоторый коллектор, следует вызвать метод collect потока:

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

В псевдоклассе Collectors представлен набор статических методов, предназначенных для создания стандартных библиотечных коллекторов:

1. **static** <T> Collector<T,?,List<T>> toList()

2. `static <T,K,U> Collector<T,?,Map<K,U>> toMap(`
   `Function<? super T,? extends K> keyMapper,`
   `Function<? super T,? extends U> valueMapper`
   `)`

3. `static <T> Collector<T,?,Set<T>> toSet()`

4. `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(`
   `Function<? super T,? extends K> classifier`
   `)`

3 Задание

В ходе выполнения лабораторной работы требуется разработать на языке Java один из статических методов, перечисленных в таблице 1. В теле метода нельзя использовать такие управляющие конструкции языка Java, как операторы выбора и циклы.

В методе main вспомогательного класса Test нужно продемонстрировать работоспособность разработанного метода.

Таблица 1: Варианты заданий

1	`static List<Integer> minimums(ArrayList<ArrayList<Integer>> a)` Формирует список минимумов из массива массивов целых чисел.
2	`static List<Integer> sums(ArrayList<ArrayList<Integer>> a)` Формирует список сумм из массива массивов целых чисел.
3	`static Map<String, Integer> maximums (HashMap<String, ArrayList<Integer>> a)` Формирует отображение строк в максимумы на основе таблицы, отображающей строки в массивы целых чисел.
4	`static Map<String, Integer> sums (HashMap<String, ArrayList<Integer>> a)` Формирует отображение строк в суммы на основе таблицы, отображающей строки в массивы целых чисел.
5	`static Map<String, Integer> inverse(HashMap<Integer, String> a)` Инвертирует отображение, заданное хеш-таблицей (если несколько целых чисел отображаются в одну и ту же строку, такая строка не включается в результирующее отображение).
6	`static Map<String, String> compose (HashMap<String, Integer> a, HashMap<Integer, String> b)` Вычисляет композицию двух отображений, заданных хеш-таблицами.
7	`static Map<String, List<Integer>> inverse (HashMap<Integer, ArrayList<String>> a)` Инвертирует отображение, заданное хеш-таблицей.
8	`static Map<Integer, Integer> selfCompose (HashMap<Integer, Integer> a)` Вычисляет композицию отображения, заданного хеш-таблицей, с самим собой.
9	`static Map<String, Set<Integer>> merge (ArrayList<HashMap<String, Integer>> a)` Выполняет слияние хеш-таблиц, отображающих строки в числа, в одно отображение строк в множества чисел.
10	`static Map<String, Integer> counts(ArrayList<String> a)` Подсчитывает, сколько каждая строка встречается в массиве.