УДК 004.43

ФУНКЦИОНАЛЬНАЯ РЕАЛИЗАЦИЯ КОНСТРУКЦИИ SWITCH-CASE В ЯЗЫКЕ JAVA 8

О. Ю. Рязанов, Ю. Д. Рязанов

Белгородский государственный технологический университет им. В. Г. Шухова

Поступила в редакцию 24.10.2017 г.

Аннотация. В статье рассмотрены недостатки оператора множественного выбора в языке *Java* 8. Предложен способ функциональной реализации нового оператора, имеющего существенные преимущества над конструкцией *switch-case* в *Java* 8. Новый оператор множественного выбора реализуются в виде библиотеки и не требует внесения изменений в компилятор языка *Java* 8.
Ключевые слова: языки программирования, функциональное программирование, объектно-ориентированное программирование, параллельное программирование.
Annotation. The article discusses the shortcomings of the multiple choice operator in the *Java* 8 language. A method for the functional implementation of a new operator with significant advantages over the *switch-case* construct in *Java* 8 is proposed. The new multiple choice operator is implemented as a library and does not require changes to the *Java* 8 compiler.
Keywords: programming languages, functional programming, object-oriented programming, parallel programming.

ВВЕДЕНИЕ

Язык *Java* создавался как объектно-ориентированный язык программирования. Продолжительное время объектно-ориентированный подход к программированию являлся главным подходом и язык *Java* приобрел большую популярность [1]. В настоящее же время функциональные языки программирования становятся все более популярными. Во многом это происходит благодаря популяризации реактивного программирования – программирования, ориентированного на события. В связи с современными тенденциями, функциональные возможности языка *Java* существенно расширились благодаря лямбда-выражениям, появившимся в *Java* 8 [2]. Так же в *Java* 8 появились *Collection* и *Stream API* для работы с ними в функциональном стиле. Использование функций из *Stream API* позволяют избегать программистам таких громоздких конструкций, как *for* и *forEach*. Эти возможности упростили читаемость кода, скорость его написания, и снизили вероятность допущения ошибок программистом.

В *Java* 8 появилась возможность работать в функциональном стиле с циклами, однако функциональной альтернативы конструкции множественного выбора *switch-case* нет. Современные подходы к архитектуре программного обеспечения практически вытеснили оператор множественного выбора из использования в коде, однако правильные архитектурные решения зачастую влекут за собой увеличение количества кода, классов и пакетов.

В этой статье рассмотрены недостатки оператора множественного выбора в *Java* 8, способы их устранения, и предложен способ функциональной реализации оператора, имеющий существенные преимущества над конструкцией *switch-case* в *Java* 8.

АНАЛИЗ КОНСТРУКЦИИ *SWITCH-CASE* В *JAVA* 8

В процессе разработки программного обеспечения на языке *Java* 8, программисты стараются избегать оператора множественного

выбора по ряду причин. Некоторые из них относятся к синтаксическим ограничениям языка, другие – к недостаткам архитектуры приложения, третьи – к проблемам распараллеливания алгоритмов. Далее приведены недостатки, выявленные в результате анализа конструкции *switch-case*.

Первым недостатком конструкции *switch-case* является то, что выбор варианта ветвления может осуществляться только по примитивам и перечислениям. В операторе *switch-case* невозможно выполнить выбор по объекту, даже не смотря на то, что в нем переопределен метод *equals* [3]. Кроме этого, ни объект, по которому осуществляется выбор, ни вариант ветвления не может быть функцией.

Иногда бывает необходимо выполнить действия в том случае, если результатом применения функции к объекту является истина. Например, если есть несколько множеств объектов и необходимо применить функцию в случае принадлежности объекта конкретному множеству. Практическим примером такой ситуации является выбор источника для получения данных: из памяти, с диска или с удаленного сервера. В случае наличия объекта в памяти, будут осуществлены действия с ним. В случае отсутствия объекта в памяти и наличии его на диске, объект загружается в память и осуществляются действия с ним. В противном случае, объект будет загружен с сервера и обработан.

В некоторых случаях удобно применить выражение *Predicate* – функциональный интерфейс, который проверяет соблюдение некоторого условия и может быть использован для лямбда-выражений [4]. Простейшим примером ситуации, в которой удобно применить условие, является проверка знака числа и вывод сообщения о его результате. Для реализации такой программы средствами языка *Java* 8 необходимо написать один оператор ветвления с двумя вариантами, в одном из которых будет еще один оператор ветвления с двумя вариантами, так как стандартный оператор *switch-case* не имеет возможностей для решения поставленной задачи.

Вторым недостатком конструкции *switch-case* является нарушение одного из принципов *SOLID* – принципа открытости-закрытости, сформулированного Бертраном Мейером [5]. Это принцип устанавливает следующее: «программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения». Если появляется необходимость добавить или удалить вариант ветвления в операторе множественного выбора, то программисту придется вносить изменения (удалять или добавлять вариант ветвления) в код готового класса. Возможности расширить класс извне, предоставить классу список вариантов и действий для оператора множественного выбора при использовании конструкции *Java* 8 нет.

Конечно, все эти недостатки решаются средствами языка *Java*, но они не всегда являются удобными. Первый описанный недостаток решается использованием нескольких конструкций *if*. С их помощью можно осуществить множественный выбор по объекту, однако объем кода значительно увеличится, что также снизит его читабельность. Помимо этого, второй описанный недостаток конструкции *switch-case* остается. Применение паттерна «компоновщик», описанного в *Design Patterns*, может помочь решить проблему отсутствия расширяемости конструкции. Для этого необходимо создать общий интерфейс и класс, для каждой альтернативы в конструкции *switch-case*. В каждом классе будет только одна конструкция *if*, в которой осуществляется сравнение текущего объекта с вариантом с помощью *equals*. При соответствии можно как прерваться и не сравнивать далее, так и передать объект для сравнения и выполнения тела другого варианта ветвления. Применение этого шаблона проектирования решает практически все проблемы конструкции *switch-case*. Но за собой он влечет ряд других. Для каждой альтернативы необходимо создать класс. Большое количество классов приведет к увеличению времени компиляции. От увеличения числа файлов пострадает и структура проекта.

ФУНКЦИОНАЛЬНАЯ РЕАЛИЗАЦИЯ КОНСТРУКЦИИ *SWITCH-CASE* И ЕЕ ПРЕИМУЩЕСТВА

Проведя анализ оператора множественного ветвления в языке *Java* 8, можно сделать вывод, что для решения некоторых специфических задач в языке программирования не хватает средств и выразительных возможностей. Решение этой задачи может заключаться в разработке нового компилятора, транслятора исходных текстов программ из расширенного языка в базовый, либо в разработке новой библиотеки в рамках существующего языка [6]. Наиболее простой задачей является разработка библиотеки, содержащей в себе функциональную реализацию конструкции *switch-case*, которая решает все описанные недостатки оператора множественного выбора в языке *Java* и имеет преимущества по сравнению с ним.

Предлагаемая авторами функциональная реализация оператора *switch-case* выполнена в соответствии с концепцией *Fluent Interface* [7]. С целью возможности осуществления этой концепции реализованы классы *Switch*, *Context* и класс *SwitchStep*. В основе реализации функционального подхода к оператору множественного выбора лежит класс *Switch*. Этот класс необходим для инстанцирования объектов класса *SwitchStep* и класса *Context* – контекста выполнения операций. Класс *Switch* содержит в себе статический метод *by*, который имеет несколько реализаций с разными входными параметрами:

1. *SwitchStep by* (*Object o*) – входным параметром является объект, по которому осуществляется ветвление.

2. *SwitchStep by* (*Predicate p*) – входным параметром является предикат, который будет применен к варианту ветвления, и в случае его истинности – выбрано ветвление.

Класс *Context* – контекст представляет собой хранилище статусов операций, событий и ошибок. Самым важным полем в контексте является объект, по которому осуществляется поиск необходимого варианта ветвления. Также в классе контекста имеется поле *selectedOption*. Оно хранит состояние выполнения ветвления. Это поле может принимать значение *false*, если еще не был осуществлен переход ни по одной ветви, или *true*, если был найден подходящий вариант. Это поле необходимо для прекращения выполнения поиска соответствующих вариантов. Помимо этого состояния, контекст хранит список ошибок, которые произошли во время выполнения. Кроме этих полей контекст содержит ряд других полей, необходимых для контроля и хранения состояния *Switch*.

Класс *SwitchStep* – является «оберткой» контекста, определяющей, какие методы могут быть далее в цепочке. В классе *SwitchStep* реализованы следующие методы:

1. *SwitchStep scase*(*Object variant, CaseAction action*) – эквивалент секции *case* без оператора *break*;

2. *SwitchStep scases*(*List<Pair> variants*) – эквивалент множества последовательных секций *case* без оператора *break*;

3. *SwitchStep scaseBreak*(*Object variant*, *CaseAction action*) – эквивалент секции *case* с оператором *break*;

4. *SwitchStep scasesBreaks*(*List<Pair> variants*) – эквивалент множества последовательных секций *case* с операторами *break* в каждой.

Все функции возвращают один и тот же объект, который был создан функцией *by*, но с измененным контекстом. Это сделано для того, чтобы следовать концепции *Fluent Interface* и вызывать цепочку методов в одном операторе.

Функции *scases* и *scasesBreaks* позволяют дополнять варианты объектов и соответствующих им действий в процессе выполнения программы. Благодаря передаче в функцию списка пар (объект-действие), можно писать классы, которые соблюдают принцип открытости-закрытости.

Ниже приведен пример кода с использованием функциональной реализации конструкции *switch-case*:

```
Switch.by(object)
    .scaseBreak(obect1, () -> {
      //do something
    })
    .scaseBreak(object2, () -> {
      //do something
```

```
})
.scaseBreak(object3, () -> {
   //do something
});
```

Как можно заметить, в результате получился удобочитаемый код, который сопоставим по объему с обычным *switch-case*. Однако функциональный подход к реализации оператора множественного выбора является более гибким. Кроме устранения недостатков стандартного оператора, функциональная реализация предоставляет ряд дополнительных возможностей. Одной из таких возможностей является удобное и расширяемое распараллеливание алгоритмов.

РАСПАРАЛЛЕЛИВАНИЕ ОПЕРАЦИЙ В ФУНКЦИОНАЛЬНОЙ РЕАЛИЗАЦИИ КОНСТРУКЦИИ *SWITCH-CASE*

Существует ряд практических задач, в которых используется множественное ветвление и есть возможность распараллелить операции в них. В некоторых из них имеется возможность выполнить блок операторов, соответствующих варианту ветвления, в других – параллельно получить значение функции, которая является вариантом ветвления. Далее такая функция будет называться функцией выбора. Примером такой задачи является приведенная выше задача выбора источника для получения данных. Определить, имеется ли объект в каком-либо источнике данных можно параллельно, так как нет необходимости в синхронизации потоков для выполнения проверок источников данных на наличие данных.

Для выполнения параллельных операций в операторе множественного выбора существует ряд методов, объявленных в классе *SwitchStep*:

1. *SwitchStep pcase*(*Object variant, CaseAction action*) – эквивалент секции *case* без оператора *break*;

2. *SwitchStep pcases*(*List<Pair> variants*) – эквивалент множества последовательных секций *case* без оператора *break*;

3. *SwitchStep pcaseBreak*(*Object variant*, *CaseAction action*) – эквивалент секции *case* с оператором *break*;

4. *SwitchStep pcasesBreaks*(*List<Pair> variants*) – эквивалент множества последовательных секций *case* с операторами *break* в каждой;

5. *void wait*() – функция, блокирующая вызывающий поток выполнения программы до окончания выполнения одного или нескольких блоков операторов ветвления.

Выполнение параллельных операций в операторе множественного выбора реализовано с использованием паттерна наблюдатель [8]. Наблюдаемым объектом является результат выполнения функции, которая представляет собой вариант ветвления. Контекст оператора множественного выбора с возможностью осуществления параллельных операций содержит в себе список наблюдаемых объектов. Функции *pcase* только добавляют в контекст пары (функция выбора – блок операций). В функции *wait* осуществляется вызов функций выбора в отдельных потоках и объявление *callback*'ов для них. *Callback* содержит в себе вызов функции, которая является блоком операторов варианта ветвления, а также изменение переменной контекста, которая является счетчиком выполненных блоков. Последний оператор функции wait – пустой цикл. Этот цикл выполняется до тех пор, пока в контексте выполнения значение счетчика выполненных блоков не будет равно количеству блоков в контексте. Важно отметить, что переменная-счетчик объявлена как *volatile* для избежания одновременного изменения переменной в разных потоках.

Аналогичным образом, с использованием паттерна наблюдатель, реализованы функции *pcaseBreak*. Различие между реализацией этих функций и функций *pcase* является структура *callback*'а. Если результатом выполнения функции выбора является истина, сообщающая о том, что необходимо выполнить соответствующий ей блок операторов, то в переменную контекста, которая содержит в себе список номеров пар (функция выбора – блок операций), функция выбора которых вернула истину, помещается номер текущей пары. Как и в функции *pcase*, увеличивается счетчик выполненных функций выбора в контексте. Если результатом выполнения является

ложь, то только увеличивается счетчик в контексте. Затем следует цикл, условие выхода из которого следующее: в контексте есть номер пары, который меньше, чем у текущей пары, или счетчик выполненных пар равен количеству пар. Далее выполняется проверка: если в контексте был номер пары с меньшим номером, чем у текущей пары, то блок операций не выполняется, иначе выполняется блок операций. После выполнения операции в контексте увеличивается значение счетчика выполненных операций.

Кроме параллельного выполнения функции выбора, параллельно могут выполняться блоки операторов. Блоки операторов выполняются параллельно, если не используется оператор *break*. Этот функционал реализован способом, аналогичным описанному выше. Отличие лишь в функции *wait*, которая блокирует поток до выполнения всех блоков, которые оказались подходящими вариантами ветвления.

ЗАКЛЮЧЕНИЕ

Представленная функциональная реализация конструкции *switch-case* позволяет использовать множественное ветвление в алгоритмах, где существует необходимость сделать выбор по объекту, оставить класс открытым для расширения. Использование такой реализации позволяет написать компактный и удобочитаемый код, который можно распараллелить. Распараллеливание кода дает выигрыш в скорости выполнения программы. Функциональная реализация оператора множественного ветвления имеет как качественные, так и количественные преимущества перед оператором в языке *Java* 8.

Немаловажным является то, что новый оператор множественного выбора реализуются без внесения изменений в компилятор языка *Java* 8.

Работа поддержана грантом РФФИ № 16-07-00487.

СПИСОК ЛИТЕРАТУРЫ

1. TIOBE Index for October 2017 – Режим доступа: https://tiobe.com/tiobe-index/

2. *Уорбэртон Р.* Лямбда-выражения в Java 8. Функциональное программирование – в массы / Р. Уорбэртон: пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2014. – 192 с.

3. Java™ Platform Standard Ed. 8 – Режим доступа: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-

4. Java™ Platform Standard Ed. 8 – Режим доступа: https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html

5. *Meyer Bertrand*. Object-Oriented Software Construction / B. Meyer. – ISE Inc., 2011. – 1406 p.

6. *Соломатин Д. И.* Реализация синтаксических расширений языков программирования на примере диалекта языка Java для параллельных вычислений / Д. И. Соломатин // Вестник Воронеж. гос. ун-та. Сер. Системный анализ и информационные технологии. – 2010. – № 2. – С. 118–124.

7. *Martin Fowler*. FluentInterface, 20 December 2005. – Режим доступа: https://martinfowler.com/bliki/FluentInterface.html

8. *Гамма Э.* Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – Санкт-Петербург : Питер, 2017. – 368 с.

Рязанов О. Ю. – студент 4 курса кафедры программного обеспечения вычислительной техники и автоматизированных систем, Белгородский государственный технологический университет им. В. Г. Шухова.
Тел.: +7 (915) 571-75-85
E-mail: weery@live.ru

Рязанов Ю. Д. – доцент кафедры программного обеспечения вычислительной техники и автоматизированных систем, Белгородский государственный технологический университет им. В. Г. Шухова.
Тел.: +7 (910) 325-73-75
E-mail: Ryazanov.iurij@yandex.ru.

Ryazanov O. Yu. – 4th Year Student of the Department of Software Computer and Automated Systems, BSTU name after V. G. Shukhov.
Tel.: +7 (915) 571 – 75 – 85
E-mail: weery@live.ru

Ryazanov Yu. D. – Associate Professor of the Department of Software Computer and Automated Systems, BSTU name after V. G. Shukhov.
Tel.: +7 (910) 325 – 73 – 75
E-mail: Ryazanov.iurij@yandex.ru