



Stage #14

Stage #14

Java Basic Syntax. One-Dimensional Arrays. Packages and Imports. MVC Pattern. Version Control Systems, VCS. Basic of git and GitHub



LEARN. GROW. SUCCEED.

© 2024. STEP Computer Academy - a leader in the field of professional computer education
by Viktor Ivanchenko / ivanvikvik@gmail.com / Minsk

Stage #14

Базовый синтаксис языка Java.

Одномерные массивы. Использование пакетов и импортов. Архитектурный шаблон проектирования MVC.

Основы управления и отслеживания версий ПО. Системы контроля (управления) версиями. Основы git и GitHub

Цель работы 

Научиться грамотно описывать пользовательские типы данных и распределять их по Java-пакетам, а также практически закрепить использованием одномерных массивов в языке Java на примере разработки интерактивных приложений с использованием архитектурного шаблона проектирования Model-View-Controller и системы контроля/управления версиями git с сохранением на облачном хостинг-сервисе GitHub.

Что нужно запомнить (краткие тезисы)

Перед написанием очередного приложения, очень важно вспомнить (запомнить) следующие вещи:

1. Менеджер исходного кода (*Source Code Manager, SCM*) или система контроля версий (**Version Control System, VCS**) – специальная утилита, которая управляет и отслеживает версии программного обеспечения.
2. **git** – мощный и гибкий инструмент управления версиями и их изменениями.
3. **git** был разработан **Линусом Торвальдом** в **2005** году для поддержки разработки ядра операционной системы **Linux**.
4. Наиболее популярная интерпретация слову *git* – *Global Information Tracker*. Сам же Линус Торвальдс использовал британский термин «*git*», на сленге означающий «гнилой человек» или «мерзавец». Почему – знает только он.

5. В языке Java **модулем компиляции** называется исходный код программы на языке Java, сохранённый в текстовом файле с расширением *.java.
6. Модуль компиляции может состоять из трёх необязательных частей: декларации пакета, секции описания импортов и описание пользовательских типов верхнего уровня.
7. В языке Java в одном модуле компиляции можно описать только один общедоступный тип (с модификатором доступа public). Остальные типы должны быть с модификатором доступа по умолчанию, который не ставится.
8. Лозунг Java-разработчиков: **один тип – один модуль компиляции**, т.е. лучше в исходном файле программы описывать только один тип.
9. **Program File Name** – имя исходного файла программы (модуля компиляции) должно в точности совпадать с именем общедоступного (открытого, public) типа, описанного в нём. Если этого не сделать, то исходный файл не будет компилироваться (пример: предположим, 'Student' – имя публичного класса, тогда имя исходного файла, в котором описан данный класс, должно быть следующим 'Student.java').
10. Если в модуле компиляции нет ни одного общедоступного типа, то имя файла исходного кода может не совпадать с именами типов, описанных в нём.
11. Пользовательские типы данных в языке Java обычно объединяются в пакеты.
12. **Пакеты (packages)** являются более крупной единицей деления кода и представляют собой набор взаимосвязанных модулей, предназначенных для решения задач определенного класса некоторой предметной (проблемной) области.
13. **Пакеты** в языке Java – это способ структуризации модулей. Пакеты призваны разбить (разделить) программу на логические независимые составляющие части (наборы), содержимое которых затем можно повторно использовать в других программах.
14. На физическом уровне пакет в языке Java – это **папка (директория)!!!**
15. Любая программа на языке Java всегда имеет минимум один пакет – это **пакет по умолчанию (default package)** – пакет, который не имеет имени.
16. Пользовательские типы данных, которые описаны в default-пакете, нельзя повторно использовать в других программах.

17. Если для пользовательского типа данных пакет не определён, то считается, что данный тип находится в пакете по умолчанию. Данный тип нельзя будет повторно использовать в других программных проектах.
18. Чтобы задекларировать пользовательский тип данных в соответствующем пакете необходимо использовать директиву ***package***, после которой указывается имя пакета:

```
package stage14;
```

19. Декларация пакета в модуле компиляции может быть только одна и всегда располагаться самой первой строчкой.
20. Пакеты могут быть вложены друг в друга. Их декларация использует точку для разделения соответствующих имён вложенных пакетов:


```
package javastages.stage14.task01;
```

21. Согласно соглашению по именованию пакетов в языке Java название пакета пишется всегда ***строчными*** (маленькими) буквами, должно однозначно отражать его содержимое и обычно называется ***именем существительным***.
22. Каждый пакет имеет своё пространство имён, что позволяет избежать конфликта имён между разрабатываемыми пользовательскими типами данных.
23. Декларирование пакета при описании типа наделяет его уникальным составным именем. Квалифицированное (полное) имя состоит из имени пакета (а также вложенных пакетов) и имени самого пользовательского типа.
24. Чтобы квалифицированное имя пользовательского типа данных было достаточно уникальным компания Sun Microsystems рекомендует в названии начальных пакетов использовать обратное доменное имя компании-разработчика. К примеру, для IT Academy Step для уникальности пользовательских типов данных можно использовать следующие начальные пакеты:



```
package by.itstep....;
```

25. Если пользовательский тип данных описан в пакете, то доступ к нему в других пакетах уже осуществляется через его полное квалифицированное имя. К примеру, если класс *Student* описывается с декларацией пакета *by.itstep.model.entity*, то доступ к нему в пакете *by.itstep.controller* должен осуществляться по следующему полному имени *by.itstep.model.entity.Student*:

```
package by.itstep.model.entity;  
public class Student {  
    //...  
}
```




```
package by.itstep.controller;  
public class Main {  
    public static void main(String[] args) {  
        by.itstep.model.entity.Student st =  
            new by.itstep.model.entity.Student();  
        //...  
    }  
}
```



26. Глубина вложенности пакетов практически не ограничивается ничем, кроме файловой системы.
27. Для упрощения использования типов, описанных в пакетах, в языке Java используются **import-выражения**.
28. В языке Java *import*-выражения позволяют при программировании разработчику указывать не полное название соответствующего типа, а его простое название (имя), а во время компиляции компилятор сам расставит нужные полные имена типов.
29. В исходном файле программы на языке Java *import*-выражения должны следовать непосредственно за оператором **package** (если таковой имеется) и перед любыми определениями пользовательского типа.
30. Чтобы обращаться по простому имени к типу *Student*, который описан в пакете *by.itstep.model.entity*, необходимо в самом начале после декларации пакета прописать *import*-выражение: *import by.itstep.model.entity.Student*. Пример импортирование имени конкретного типа:

```
package by.itstep.controller;  
import by.itstep.model.entity.Student;  
  
public class Main {  
    public static void main(String[] args) {  
        Student st = new Student();  
    }  
}
```




```
    //...  
  }  
}
```

31. Чтобы обращаться по простому имени ко всем типам целевого пакет *by.itstep.model.entity*, необходимо в самом начале после декларации пакета прописать *import*-выражение со звёздочкой в конце вместо имени типа: *import by.itstep.model.entity.**. Пример импортирование всех имён соответствующего пакета:

```
package by.itstep.model.entity;  
public class Student {  
    //...  
}
```

```
package by.itstep.model.entity;  
public class Group {  
    //...  
}
```

```
package by.itstep.controller;  
import by.itstep.model.entity.*;  
  
public class Main {  
    public static void main(String[] args) {  
        Student st = new Student();  
        Group group = new Group();  
        //...  
    }  
}
```



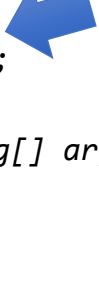
32. Стандартный пакет ***java.lang***, в котором находятся все критические для любой Java-программы типы (*Object*, *String*, *Integer*, *System* и т.д.), импортируется в каждый модуль компиляции автоматически. Следовательно, ко всем типам из данного пакета можно сразу обращаться по простому имени без использования *import*-выражений.
33. В языке Java есть также особая форма импорта – статический импорт. С помощью ключевых слов ***import*** и ***static*** можно организовать доступ по простому

имени к статическим компонентам типа (класса, перечисления или интерфейса). Под статическим компонентом типа подразумевается статическое поле или статический метод.

34. Пример доступа к конкретным статическим компонентам *E* и *PI* класса *Math* из пакета *java.lang*:

```
import static java.Lang.Math.E;
import static java.Lang.Math.PI;

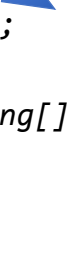
public class Main {
    public static void main(String[] args) {
        double eValue = E;
        double piValue = PI;
        //...
    }
}
```



35. Пример доступа ко всем статическим компонентам класса *Math* из пакета *java.lang*:

```
import static java.Lang.Math.*;

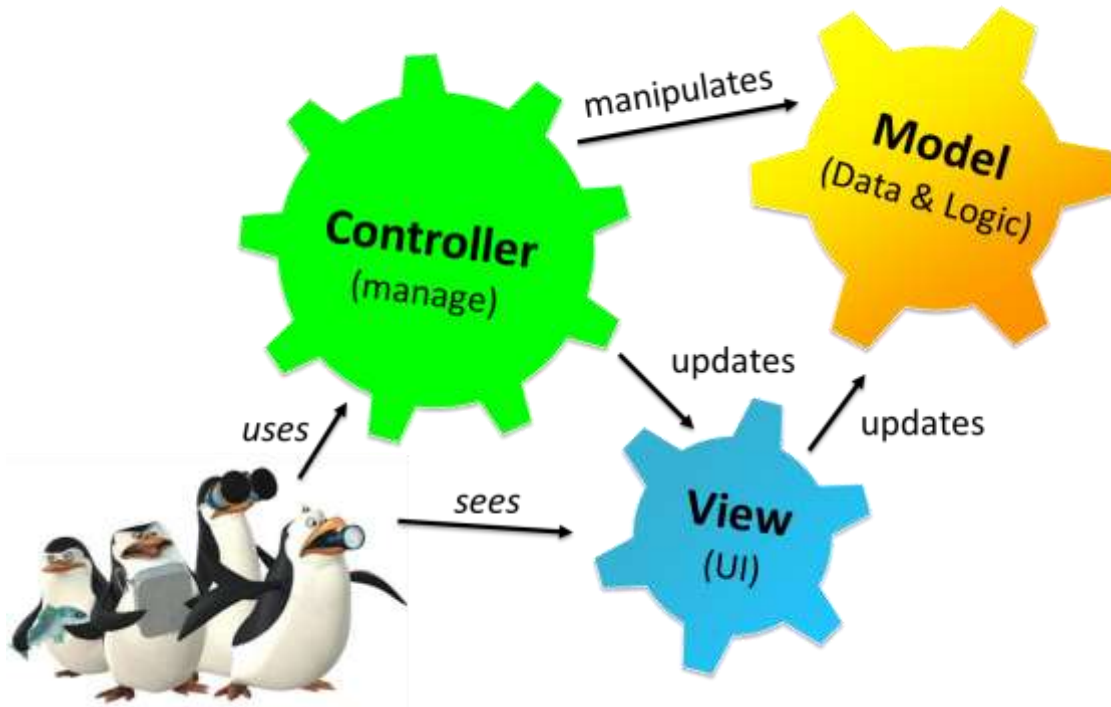
public class Main {
    public static void main(String[] args) {
        double eValue = E;
        double piValue = PI;
        //...
    }
}
```



36. Разработка программы как совокупность пакетов пользовательских типов позволяет:

- упростить задачи проектирования программы и распределения процесса разработки между разработчиками;
- изолировать различные типы друг от друга;
- повторно использовать описанные пользовательские типы пакета в других проектах;
- задать пользовательских типам данных пакетов более уникальные имена;
- предоставить дополнительную реализацию инкапсуляции для пользовательских типов данных на уровне пакетов;
- предоставить возможность лёгкого и безопасного обновления (замены), без необходимости изменения остальной части системы;
- упростить тестирование программы;

- упростить процесс обнаружения и исправления ошибок.
37. Шаблон **Model-View-Controller (MVC)** – это архитектурный шаблон проектирования. Он разделяет инфраструктуру приложения на три отдельные части, помогая организовать код по логическим представлениям в зависимости от их назначения.
38. Концепция шаблона MVC позволяет разделить данные, представление и обработку действий пользователя на три отдельных компонента:
- **модель (Model)**: предоставляет знания: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти знания можно визуализировать;
 - **представление** или **вид (View)**: отвечает за отображение информации (визуализацию);
 - **контроллер (Controller)**: обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует **модель** и **представление** для реализации необходимой реакции.



39. Как **представление**, так и **контроллер** зависят от **модели**. Однако **модель** не зависит ни от **представления**, ни от **контроллера**. Тем самым достигается назначение такого разделения: оно позволяет строить **модель** независимо от **визуального представления**, а также создавать несколько различных **представлений** для одной **модели**.

40. Основная цель применения шаблона MVC состоит в разделении бизнес-логики (**модели**) от её визуализации (**представления, вида**). За счет такого разделения повышается возможность повторного использования. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения. На пример:







- к одной **модели** можно присоединить несколько **видов**, при этом не затрагивая реализацию самой **модели**. Например, некоторые данные могут быть одновременно представлены в виде электронной таблицы, гистограммы и круговой диаграммы;
- не затрагивая реализацию **видов**, можно изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных), для этого достаточно использовать другой **контроллер**;
- ряд разработчиков специализируется только в одной из областей: либо разрабатывают графический интерфейс, либо разрабатывают бизнес-логику. Поэтому возможно добиться того, что программисты, занимающиеся разработкой бизнес-логики (**модели**), вообще не будут осведомлены о том, какое **представление** будет использоваться.

Графическое представление алгоритмов



Для общего представления решения задачи без привязки к конкретному языку программирования на практике используют блок-схемы. Они позволяют в графическом виде представить алгоритм решения задачи, который понятен не только разработчику, но даже домохозяйке.

Для графического представления алгоритмов решения задачи использую специальные унифицированные блоки, каждый из которых несёт в себе определённую смысловую нагрузку. Кратко каждый из наиболее востребованных блоков описывается в нижеприведённой таблице.

Таблица 1 – Наиболее часто используемые блоки

#	Shape (блок)	Description (описание)
1.		Блок начала/окончания выполнения программы
2.		Блок данных – используется для ввода, объявления и инициализации переменных программы
3.		Блок действия – используется для вычисления любых выражений программы
4.		Блок вызова процедур или функций – используется для обозначения вызова пользовательской функции или процедуры, код или реализация которой находится в другом файле
5.		Блок условия – задаёт соответствующие условия дальнейшего выбора хода выполнения кода программы
6.		Блок вывода данных – используется для обозначения выводимых данных или результата работы программы

Продолжение таблицы 1

7.		Блок соединитель на странице – используется в случае, если блок-схема алгоритма не может идти всё время сверху вниз и требуется её перенести на другую часть свободного места на том же листе
8.		Блок соединитель между страницами– используется в случае, если блок-схема алгоритма не помещается на одной странице и одну из её частей нужно перенести на другую страницу

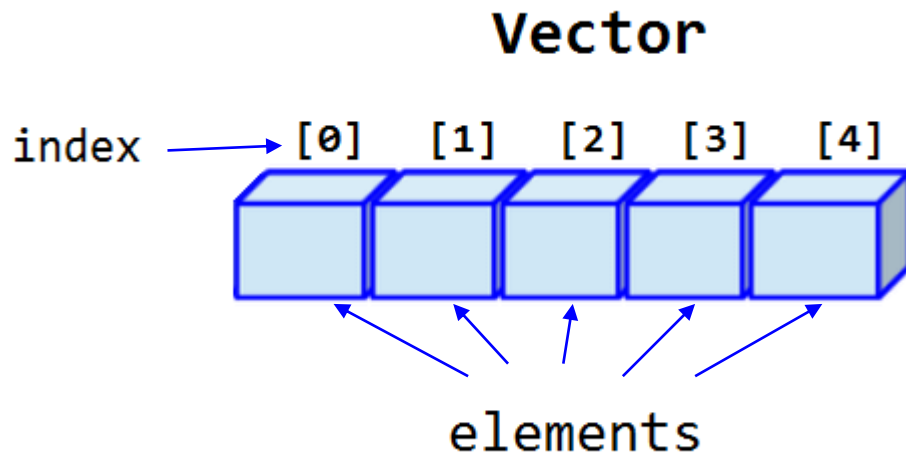
Требования

- 1) Необходимо выполнить из каждого раздела минимум по два любых задания.
- 2) Для каждого задания в начале рекомендуется разработать блок-схему алгоритма решения.
- 3) Проект обязательно должен быть сразу реализован и сохранён под управление системой контроля версий (VCS) **git** и в последующем залит в централизованный репозиторий на облачном хостинг-сервисе **GitHub**.
- 4) Создаваемые классы должны иметь «адекватные» имена и грамотно разложены по соответствующим пакетам, которые также должны иметь «адекватные» названия и начинаться с: **by.itstep.NameOfStudent**. ...
- 5) Если логически не подразумевается или в задании иного не указано, то входными и выходными данными являются вещественные числа.
- 6) В соответствующих компонентах бизнес-логики необходимо предусмотреть «защиту от дурака», т.е. прежде чем выполнять действия с данными нужно проверить, являются ли данные адекватными (непротиворечивыми).
- 7) При разработке программ рекомендуется придерживаться принципа единственной ответственности (**Single Responsibility Principle, SRP**), т.е. любой код (метод, класс и т.д.) должен быть настолько самодостаточным, насколько это возможно, чтобы его можно было повторно использовать в дальнейшем в других приложениях.
- 8) Целевые данные программы должны задаваться пользователем с консоли или генерироваться с использованием ГПСЧ (генератора псевдослучайных чисел).
- 9) Для осуществления ввода данных с консоли рекомендуется использовать соответствующие методы объекта, созданного на базе утилитного класса **java.util.Scanner**, а для вывода данных на стандартную консоль – соответствующие методы объекта **System.out**.
- 10) Для генерирования псевдослучайных данных рекомендуется использовать соответствующие методы объекта класса **java.util.Random**.
- 11) Программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом для взаимодействия с пользователем. Рекомендуется отображать интерфейс программы на английском языке.

- 12) При проверки работоспособности приложения необходимо проверить все тестовые случаи.
- 13) При разработке программ придерживайтесь соглашений по написанию кода на JAVA (*Java Code-Convention*).

Индивидуальное задание (по вариантам)

Вектор – одно из математических понятий, обозначающее последовательность однородных (однотипных) значений. Во многих языках программирования вектор эмулируется (реализуется) с помощью встроенных в язык одномерных массивов или списков.



Уровень (Level) A

- 1) Необходимо написать программу, которая подсчитывает количество ненулевых значений элементов вектора целочисленных значений.
- 2) Необходимо написать программу, которая подсчитывает количество чётных значений элементов вектора целочисленных значений.
- 3) Необходимо написать программу, которая подсчитывает количество положительных значений элементов вектора целочисленных значений.
- 4) Необходимо написать программу, которая подсчитывает количество нулевых значений элементов вектора целочисленных значений.
- 5) Необходимо написать программу, которая подсчитывает количество нечётных значений элементов вектора целочисленных значений.
- 6) Необходимо написать программу, которая подсчитывает количество отрицательных значений элементов вектора целочисленных значений.

Level B

- 1) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые больше заданного числа N , где N задаётся пользователем.
- 2) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые меньше заданного числа N , где N задаётся пользователем.
- 3) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые равны заданному числу N , где N задаётся пользователем.
- 4) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые не равны заданному числу N , где N задаётся пользователем.
- 5) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые кратны заданному числу N , где N задаётся пользователем.
- 6) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые не кратны заданному числу N , где N задаётся пользователем.
- 7) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые по модулю (абсолютному значению) больше заданного числа N , где N задаётся пользователем.
- 8) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые по модулю (абсолютному значению) меньше заданного числа N , где N задаётся пользователем.
- 9) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые по модулю (абсолютному значению) равны заданному числу N , где N задаётся пользователем.

Level C

- 1) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые больше среднеарифметического значения всех значений элементов искомого вектора.
- 2) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые больше среднегеометрического значения всех значений элементов искомого вектора.
- 3) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые меньше среднеарифметического значения всех значений элементов искомого вектора.
- 4) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые меньше среднегеометрического значения всех значений элементов искомого вектора.
- 5) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые приблизительно равны среднеарифметическому значению всех значений элементов искомого вектора с погрешностью плюс/минус D , где D задаётся пользователем.
- 6) Необходимо написать программу, которая подсчитывает количество элементов вектора целочисленных значений, которые приблизительно равны среднегеометрическому значению всех значений элементов искомого вектора с погрешностью плюс/минус D , где D задаётся пользователем.

Level D

- 1) В векторе, состоящем из вещественных элементов, вычислить произведение элементов, расположенных между максимальным и минимальным элементами.
- 2) В векторе, состоящем из целых элементов, вычислить сумму элементов, расположенных между первым и последним нулевыми элементами.
- 3) В векторе, состоящем из вещественных элементов, вычислить сумму элементов, расположенных до последнего положительного элемента.
- 4) В векторе, состоящем из целых элементов, вычислить произведение элементов, расположенных между первым и вторым нулевыми элементами.

- 5) В векторе, состоящем из вещественных элементов, вычислить элементов, расположенных между первым и вторым положительными элементами.
- 6) В векторе, состоящем из вещественных элементов, вычислить сумму элементов, расположенных после первого положительного элемента.
- 7) В векторе, состоящем из вещественных элементов, вычислить произведение элементов, расположенных после максимального по модулю элемента.
- 8) В векторе, состоящем из целых элементов, вычислить сумму элементов вектора, расположенных после последнего элемента, равного нулю.
- 9) В векторе, состоящем из вещественных элементов, вычислить сумму положительных элементов, расположенных до максимального элемента.
- 10) В векторе, состоящем из вещественных элементов, вычислить сумму элементов, расположенных между первым и последним положительными элементами.
- 11) В векторе, состоящем из вещественных элементов, вычислить сумму элементов, расположенных после максимального элемента.
- 12) В векторе, состоящем из вещественных элементов, вычислить сумму элементов, расположенных до минимального элемента.
- 13) В векторе, состоящем из вещественных элементов, вычислить сумму элементов, расположенных между первым и последним отрицательными элементами.
- 14) В векторе, состоящем из вещественных элементов, вычислить сумму элементов, расположенных между первым и вторым отрицательными элементами.
- 15) В векторе, состоящем из вещественных элементов, вычислить сумму элементов, расположенных после минимального элемента.

Best of LUCK with it, and remember to HAVE FUN while you're learning :)

Victor Ivanchenko



Пример выполнения задания с использованием архитектурного шаблона проектирования MVC

Задание

Разработайте программу нахождения периметра и площади квадрата с заданной стороной a при условии, что пользователь вводит «адекватные» данные.

Решение

- 1) Перед написанием основного приложения разработаем алгоритм решения с учётом того, что площадь квадрата вычисляется по формуле $S = a * a$, а его периметр – $P = 4 * a$:

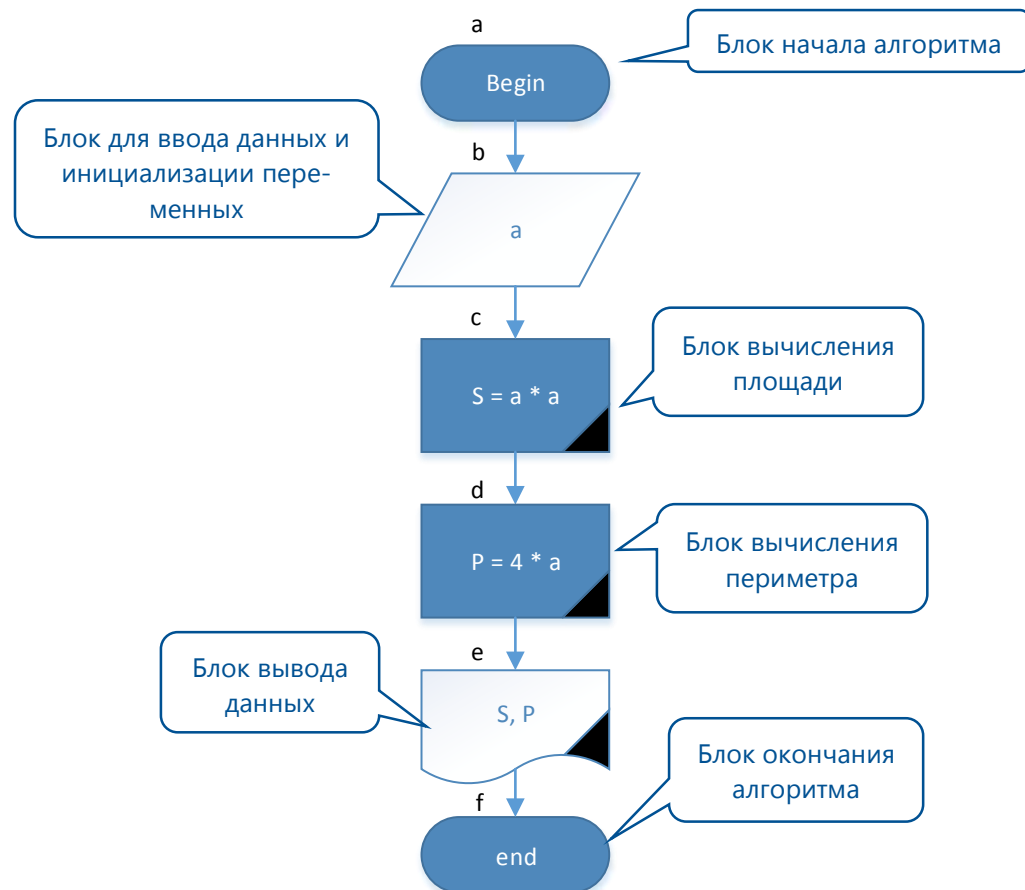


Рисунок 1 – Блок-схема алгоритма решения задания

Вербальное описание алгоритма (словесная последовательность действий) состоит из следующих шагов:

- а) начало выполнения алгоритма (вызов соответствующего метода);

- b) ввод (передача) значения стороны квадрата a ;
- c) нахождения площади квадрата;
- d) нахождения периметра квадрата;
- e) вывод (возврат) результат;
- f) завершение выполнения алгоритма.

Блок-схема работы алгоритма решения задачи была построена в Microsoft Office Visio и приведена на рисунке 1 выше.

2) Теперь реализуем бизнес-логику программы согласно разработанному алгоритму (составная часть модели согласно шаблону MVC) с использованием соответствующих статических методов `calculatePerimeter(...)` и `calculateArea(...)` функционального класса `SquareCalculator`, который будет описан в пакете `by.itstep.vikvik.javastages.stage14.exampletask.model.logic`. Статические методы будут принимать на вход по одному параметру типа `double` – значение стороны квадрата и возвращать значение типа `double` – соответствующий результат нахождения периметра и площади квадрата:

```
package by.itstep.vikvik.javastages.stage14.exampletask.model.logic;

public class SquareCalculator {
    public static double calculatePerimeter(double a) {
        double perimeter = 4 * a;
        return perimeter;
    }

    public static double calculateArea(double a) {
        double area = a * a;
        return area;
    }
}
```



Обратите внимание, как **приятно читать** и **сопровождать** вышеописанный код. Рекомендуется следовать такому же стилю написания программного кода

3) Разберём более подробно код основного класса бизнес-логики:

Декларируем соответствующий Java-пакет

```
package by.itstep.vikvik.javastages.stage14.exampletask.model.logic;
```

Объявление статического метода

Формальный параметр метода

```
public class SquareCalculator {
    public static double calculatePerimeter(double a) {
```

```

    double perimeter = 4 * a;
    return perimeter;
}

```

Вычисляемое выражение (периметр)

С помощью оператора **return** осуществляется немедленное прекращение выполнения метода и возврат результирующего значения

```

public static double calculateArea(double a) {
    double area = a * a;
    return area;
}

```

Вычисляемое выражение (площадь)

Возвращаемый результат

- 4) Далее реализуем в пакете *by.itstep.vikvik.javastages.stage14.exampletask.util* утилитный (вспомогательный) функциональный класс *UserInput* для пользовательского ввода данных, который для своей работы будет использовать метод **nextDouble()** объекта утилитного класса **Scanner** для ввода значения типа **double**. У класса будет единственный статический метод *input(...)*, который на вход будет принимать параметр в виде строки и возвращать значение типа *double*:

```

package by.itstep.vikvik.javastages.stage14.exampletask.util;

import java.util.Scanner;

public class UserInput {

    private static final Scanner SCANNER = new Scanner(System.in);

    public static double input(String msg) {

        System.out.print(msg);
        double value = SCANNER.nextDouble();

        return value;
    }
}

```

- 5) Разберём более подробно код утилитного класса *UserInput*:

Декларируем соответствующий Java-пакет

```

package by.itstep.vikvik.javastages.stage14.exampletask.util;

```

Импортируем имя класса **Scanner** для того, чтобы в программе можно было к нему обращаться по простому имени

```

import java.util.Scanner;

```

```

public class UserInput {

    private static final Scanner SCANNER = new Scanner(System.in);

    public static double input(String msg) {

        System.out.print(msg);
        double value = SCANNER.nextDouble();

        return value;

    }
}

```

Ожидаем ввод пользователем вещественного значения

Возвращаемый результат

Объявляем статическую константную ссылочную переменную-поле и присваиваем ей адрес на создаваемый оператором **new** объект класса **Scanner**

6) Один из последних классов, который необходимо реализовать, это класс отображения данных – компонент *View* согласно архитектурному шаблону MVC. Для этого объявим в пакете *by.itstep.vikvik.javastages.stae14.exampletask.view* класс *Printer*, у которого будет единственный статический метод *print(...)*. На вход данный метод принимает строку и ничего не возвращает, т.к. его ответственность – это создание инфраструктуры для вывода данных на консоль.

```
package by.itstep.vikvik.javastages.stage14.exampletask.view;
```

```

public class Printer {
    public static void print(String msg) {
        System.out.println(msg);
    }
}

```

Вывод содержимого параметра на системную консоль

7) На заключительном этапе соберём из разработанных компонентов (классов) готовую программу. Для этого опишем в соответствующем пакете класс *Main*, который будет выполнять роль контроллера согласно архитектурному шаблону MVC. В нём будет помещён стартовый статический метод *main(...)*:

```

package by.itstep.vikvik.javastages.stage14.exampletask.controller;

import by.itstep.vikvik.javastages.stage14.exampletask.model.Logic.SquareCalculator;
import by.itstep.vikvik.javastages.stage14.exampletask.util.UserInput;
import by.itstep.vikvik.javastages.stage14.exampletask.view.Printer;

public class Main {
    public static void main(String[] args) {
        double a = UserInput.input("Input a: ");
    }
}

```

```

        double s = SquareCalculator.calculateArea(a);

        double p = SquareCalculator.calculatePerimeter(a);

        Printer.print("Square with a = " + a + ": ");
        Printer.print("\nS = " + s);
        Printer.print("\nP = " + p);
    }
}

```

8) Разберём более подробно код стартового тестового класса:

Декларируем соответствующий Java-пакет

```
package by.itstep.vikvik.javastages.stage14.exampletask.controller;
```

Импортируем соответствующие имена классов для того, чтобы в программе можно было к ним обращаться по простому имени

```
import by.itstep.vikvik.javastages.stage14.exampletask.model.Logic.SquareCalculator;
import by.itstep.vikvik.javastages.stage14.exampletask.util.UserInput;
import by.itstep.vikvik.javastages.stage14.exampletask.view.Printer;
```

```
public class Main {
```

Метод main – стартовая точка доступа в программу

```
    public static void main(String[] args) {
        double a = UserInput.input("Input a: ");
```

Объявление вещественной переменной (сторона квадрата) и производим её инициализацию с помощью статического метода вспомогательного класса

Объявление переменной, которая будет содержать найденную площадь квадрата, заданного стороной *a*

Вызов статического метода класса *SquareCalculator*

Фактический параметр (значение), который передаётся в метод

```
        double s = SquareCalculator.calculateArea(a);
```

```
        double p = SquareCalculator.calculatePerimeter(a);
```

Объявление переменной, которая будет содержать найденный периметр квадрата, заданного стороной *a*

Вызов ещё одного статического метода класса *SquareCalculator*

Фактический параметр (значение), который передаётся в метод

```

        Printer.print("Square with a = " + a + ": ");
        Printer.print("\nS = " + s);
        Printer.print("\nP = " + p);
    }
}

```

Вывод соответствующего результата в системную консоль (терминал) с использованием пользовательского компонента *Printer*. Знак плюс внутри методов используется для осуществления преобразования значений в строку и последующей конкатенации в результирующую строку

9) В общем виде архитектура приложения представлена ниже на рисунке:

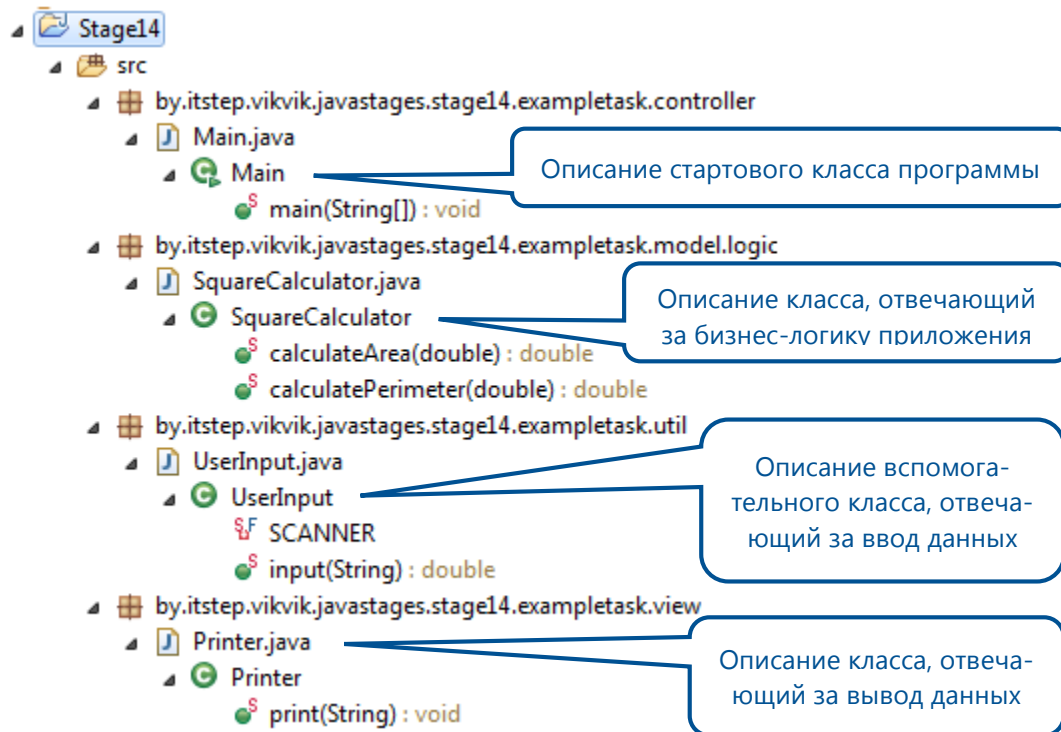


Рисунок 2 – Архитектура разработанной программы

Программа была написана и тестировалась с использованием среды разработки Eclipse Java EE IDE for Web Developers.

Компиляция программы и тестирование всех случаев (условий) выполнения с использованием инструментария JDK представлены на соответствующих рисунках 3 и 4, а их аналогичная компиляция и запуск в интегрированной среде Eclipse IDE – на рисунках 5 и 6.

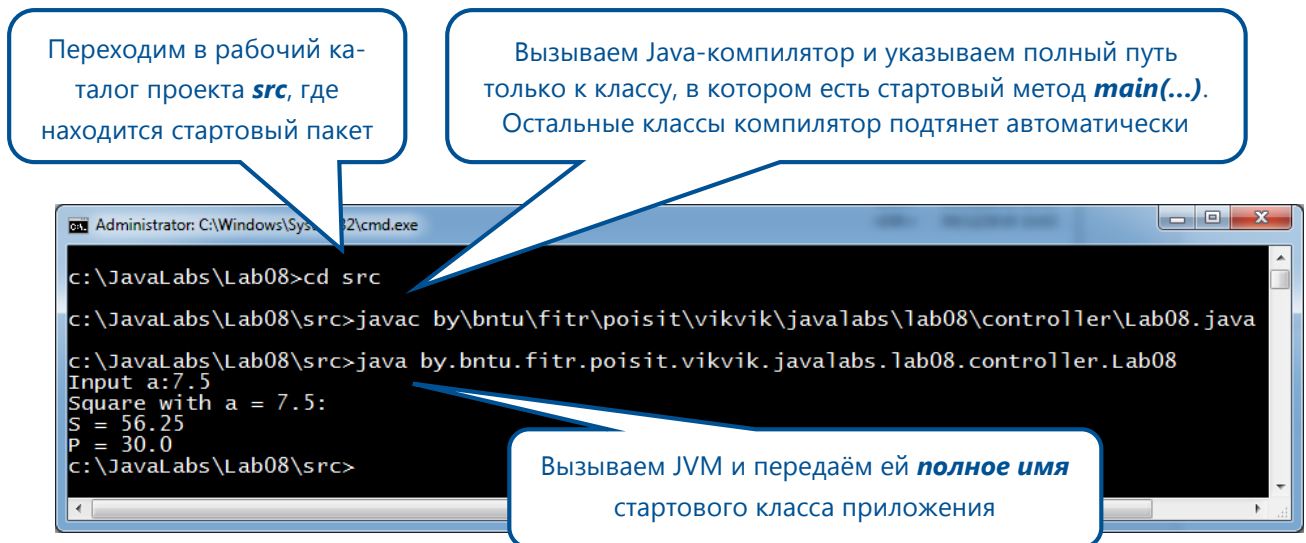


Рисунок 3 – Компиляция и запуск приложения при a = 7.5

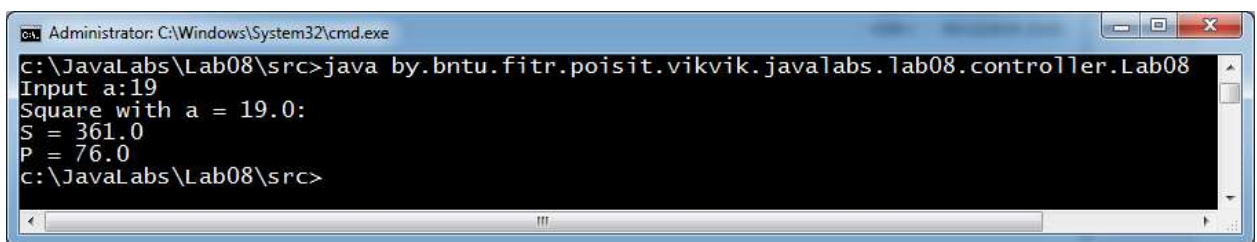


Рисунок 4 – Компиляция и запуск приложения при a = 19

```
Input a: 7.5
Square with a = 7.5:
S = 56.25
P = 30.0
```

Рисунок 5 – Компиляция и запуск приложения в среде Eclipse IDE, где a = 7.5

```
Input a: 21
Square with a = 21.0:
S = 441.0
P = 84.0
```

Рисунок 6 – Компиляция и запуск приложения в среде Eclipse IDE, где a = 21

Контрольные вопросы

- 1) Что такое пакет? Верно ли что пакет, по сути, представляет собой контейнер для типов?
- 2) Где физически храниться содержимое пакетов?
- 3) Опишите правила объявления пакетов. Какие рекомендации дают Java-разработчики по именованию пакетов?
- 4) Что такое глобальное пространство имён?
- 5) Почему важна возможность разбиения пространства имён на части с помощью объявления пакетов?
- 6) Как можно обратиться из текущего пакета к классам, которые описаны в других пакетах?
- 7) Зачем нужен пакет по умолчанию (*default package*) и какая у него есть особенность?
- 8) В чём смысл `import`-выражений? Можно ли в языке Java обойтись без секции импорта?
- 9) Какие бывают разновидности импорта в языке Java?
- 10) Как обращаться по простому имени к классу из другого пакета?
- 11) Как импортировать все классы из пакета?
- 12) Какой стандартный пакет неявно импортируется по умолчанию в любую *Java*-программу?
- 13) Какой применяется алгоритм Java-компилятором при поиске и разрешении типов для ссылочных переменных?
- 14) Что такое статический импорт и как его использовать?
- 15) Для чего нужна переменная `CLASSPATH`?
- 16) Как скомпилировать проект, который использует классы из нестандартных пакетов (точка компиляции – текущий каталог проекта *src*)?
- 17) Как скомпилировать проект, который использует классы из нестандартных пакетов (точка компиляции – любой каталог)?

- 18) Что такое code-convention?
- 19) Как необходимо правильно именовать имена идентификаторов (параметры и локальные переменные, поля и методы класса, классы, пакеты и т.д.)?
- 20) Опишите архитектурный шаблон проектирования MVC: основная цель, концепция и компоненты шаблона - модель (*Model*), представление (*View*), контроллер (*Controller*).