



Laboratory Work #23

# Java Object-Oriented Programming. Encapsulation



**LEARN. GROW. SUCCEED.**

© 2020-2021. Department: <Software of Information Systems and Technologies>  
Faculty of Information Technology and Robotics  
Belarusian National Technical University  
by Viktor Ivanchenko / [ivanvikvik@bntu.by](mailto:ivanvikvik@bntu.by) / Minsk

# ЛАБОРАТОРНАЯ РАБОТА #23

## ООП в Java. Инкапсуляция

### Цель работы

Углубить свои фундаментальные знания в использовании методологии ООП, а также научиться практически применять инкапсуляцию с использованием средств, которые предоставляет язык Java.

### Требования

- 1) Необходимо скорректировать UML-диаграмму взаимодействия классов и объектов программной системы с учётом вносимых дополнений и изменений.
- 2) При корректировке программной системы необходимо полностью использовать своё ООП-воображение и по максимум использовать возможности, которые предоставляет язык Java для программирования с использованием методологии ООП.
- 3) Необходимо скрыть реализацию компонентов и структур хранения данных с помощью грамотного применения инкапсуляции.
- 4) Каждый пользовательский тип должен иметь адекватное осмысленное имя и информативный состав (соответствующие конструкторы: по умолчанию, с параметрами, конструктор-копирования; **get**- и **set**-методы для доступа к состоянию объекта; корректно переопределённые методы базового класса *Object*: **toString()**, **equals()**, **hashCode()** и др.).
- 5) Также рекомендуется придерживаться **Single Responsibility Principle, SRP** (принципа единственной ответственности): у каждого пакета, класса или метода должна быть только одна ответственность (цель), т.е. должна быть только одна причина изменить в дальнейшем соответствующий блок кода.
- 6) Добавляемые классы необходимо грамотно разложить по соответствующим пакетам, которые должны иметь «адекватные» названия и быть вложены в указанные стартовые пакеты: ***by.bntu.fitr.poisit.nameofstudent.nameofproject***.

- 7) В соответствующих важных компонентах программной системы необходимо предусмотреть «защиту от дурака».
- 8) Для контейнерных классов в качестве хранилища данных использовать Java-массивы!!!
- 9) Все контейнерные классы должны поддерживать расширяемость, т.е. они не ограничены в объёме хранимых данных и динамически должны расширяться при их использовании в программе.
- 10) При разработке программ придерживайтесь соглашений по написанию кода на *Java* (***Java Code-Convention***) !!!

## Основное задание



Необходимо в проект, который был спроектирован и разработан в предыдущей лабораторной работе, внести следующие изменения и дополнения:

- скрыть реализацию всех компонентов и структур данных проекта, т.е. инкапсулировать все поля классов и методы, которые предназначены для внутреннего использования, с использованием модификаторов доступа языка Java, и предоставить только интерфейсную часть для внешнего взаимодействия;
- ввести, где это необходимо, высокоуровневые объекты-контейнеры, которые инкапсулируют структуру хранения множества объектов предметной области;
- убрать из класса-контроллера код по инициализации объектов предметной области и ввести соответствующие программные компоненты, которые и будут предназначены для создания и инициализации объектов предметной области, т.е. использовать компоненты в виде фабрик или строителей («креаторов»).

## Что нужно запомнить (краткие тезисы)

1. Кратко: **инкапсуляция** – сокрытие реализации за интерфейсом.
2. При использовании инкапсуляции код делят минимум на две части: **интерфейс** и **реализацию**.
3. Под **реализацией** (скрытая часть) понимается отдельные детали или полностью внутреннее устройство программного объекта (компонента, слоя, модуля, системы, ...).
4. Под **интерфейсом** (видимая часть) объекта (компонента, слоя, модуля, системы, ...) понимается то, что видно внешним по отношению к нему объектам или пользователям.
5. Полная определение относительно программного объекта: **инкапсуляция** – сокрытие отдельных (или всех) деталей внутренней реализации объекта от внешнего использования и предоставление только интерфейса для взаимодействия с ним (объектом).
6. Согласно инкапсуляции объект (компонент, слой, модуль, система, ...) должен рассматриваться как «**чёрный ящик**» - внешний объект (пользователь) не знает детали реализации и работает с ним только через интерфейс, который предоставляет данный объект.

# Графическое представление элементов

## UML-диаграммы классов

**UML** – унифицированный язык моделирования (***Unified Modeling Language***) – это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования. Его можно использовать для **визуализации, спецификации, конструирования** и **документирования** программных систем. Язык UML применяется не только для проектирования, но и с целью документирования, а также эскизирования проекта.

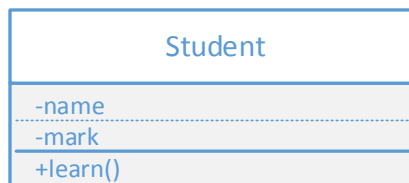
Словарь UML включает три вида строительных блоков: **диаграммы, сущности** и **связи**. **Сущности** – это абстракции, которые являются основными элементами модели, **связи** соединяют их между собой, а **диаграммы** группируют представляющие интерес наборы сущностей.

Диаграмма – это графическое представление набора элементов, чаще всего изображенного в виде связанного графа вершин (сущностей) и путей (связей). Язык UML включает **13** видов диаграмм, среди которых на первом (центральном) месте в списке – диаграмма классов.

**UML-диаграмма классов** (*Static Structure Diagram*) – диаграмма статического представления системы, демонстрирующая классы (и другие сущности) системы, их атрибуты, методы и взаимосвязи между ними.

Диаграммы классов оперируют тремя видами сущностей UML: структурные, поведенческие и аннотирующие.

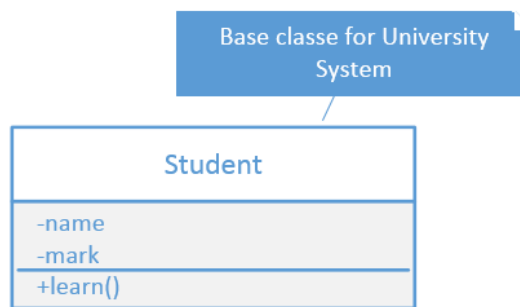
**Структурные сущности** – это «имена существительные» в модели UML. В основном, статические части модели, представляющие либо концептуальные, либо физические элементы. Основным видом структурной сущности в диаграммах классов является класс. Пример класса Студент (*Student*) с полями и методами:



**Поведенческие сущности** – динамические части моделей UML. Это «глаголы» моделей, представляющие поведение модели во времени и пространстве. Основной из них является взаимодействие – поведение, которое заключается в обмене сообщениями между наборами объектов или ролей в определенном контексте для достижения некоторой цели. Сообщение изображается в виде линии со стрелкой:



**Аннотирующие сущности** – это поясняющие части UML-моделей, иными словами, комментарии, которые можно применить для описания, выделения и пояснения любого элемента модели. Главная из аннотирующих сущностей – примечание. Это символ, служащий для описания ограничений и комментариев, относящихся к элементу либо набору элементов. Графически представлен прямоугольником с загнутым углом; внутри помещается текстовый или графический комментарий.



Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями: имя класса, атрибуты (свойства) класса и операции (методы) класса.

Для атрибутов и операций может быть указан один из нескольких типов видимости:

- + открытый, публичный (*public*)
- закрытый, приватный (*private*)
- # защищённый (*protected*)
- / производный (*derived*) (может быть совмещён с другими)
- ~ пакет (*package*)

Видимость для полей и методов указывается в виде левого символа в строке с именем соответствующего элемента.

Каждый класс должен обладать именем, отличающим его от других классов. **Имя** – это текстовая строка. Имя класса может состоять из любого числа букв, цифр

и знаков препинания (за исключением двоеточия и точки) и может записываться в несколько строк. Каждое слово в имени класса традиционно пишут с заглавной буквы (верблюжья нотация), например Датчик (*Sensor*) или ДатчикТемпературы (*TemperatureSensor*).

Для **абстрактного класса** имя класса записывается **курсивом**.

**Атрибут** (свойство) – это именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым.

Атрибут представляет некоторое свойство моделируемой сущности, которым обладают все объекты данного класса. Имя атрибута, как и имя класса, может представлять собой текст. Можно уточнить спецификацию атрибута, указав его тип, кратность (если атрибут представляет собой массив некоторых значений) и начальное значение по умолчанию.

**Статические атрибуты** класса обозначаются **подчеркиванием**.

**Операция** (метод) – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты.

Графически операции представлены в нижнем блоке описания класса.

Допускается указание только имен операций. Имя операции, как и имя класса, должно представлять собой текст. Каждое слово в имени операции пишется с заглавной буквы, за исключением первого, например move (переместить) или isEmpty (проверка на пустоту).

Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения.

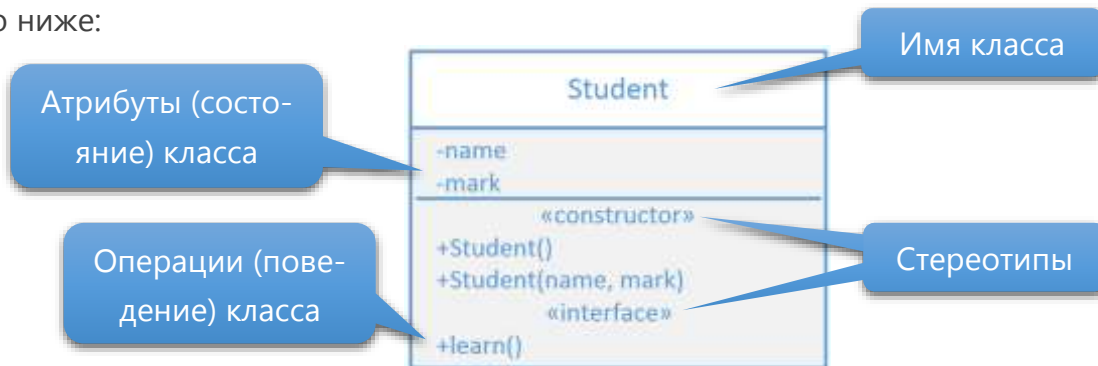
**Абстрактные методы** класса обозначаются **курсивным** шрифтом.

**Статические методы** класса обозначаются **подчеркиванием**.

Чтобы легче воспринимать длинные списки атрибутов и операций, желательно снабдить префиксом (именем стереотипа) каждую категорию в них. В данном случае **стереотип** – это слово, заключенное в угловые кавычки, которое указывает то, что за ним следует.



Общее описание класса с именем Student и другими атрибутами представлено ниже:



Существуют следующие типы связей в UML: **зависимость**, **ассоциация** (и её разновидности: **агрегация** и **композиция**), **наследование** (обобщение) и **реализация**. Эти связи представляют собой базовые строительные блоки для описания отношений в UML, используемые для разработки хорошо согласованных моделей.

Первая из них – **зависимость** – семантически представляет собой связь между двумя элементами модели, в которой *изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого)*. Графически представлена пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит еще одна; может быть снабжена меткой.



Зависимость – это связь *использования*, указывающая, что изменение спецификаций одной сущности может повлиять на другие сущности, использующие её.

**Ассоциация** – это структурная связь между элементами модели, которая описывает набор связей, существующих между объектами. Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому. Например, класс Человек (*Human*) и класс Школа (*School*) имеют ассоциацию, так как человек может учиться в школе. Ассоциации можно присвоить имя «учится в». В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации.

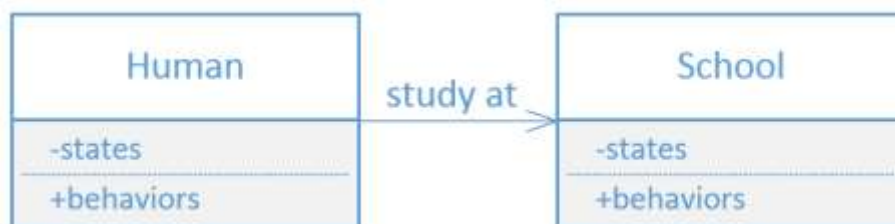
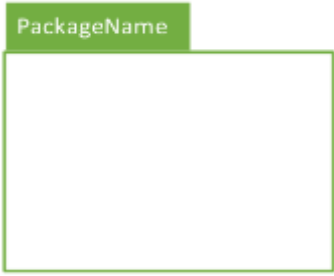
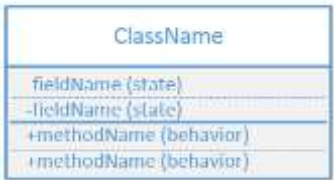
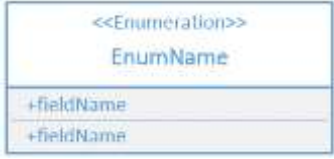





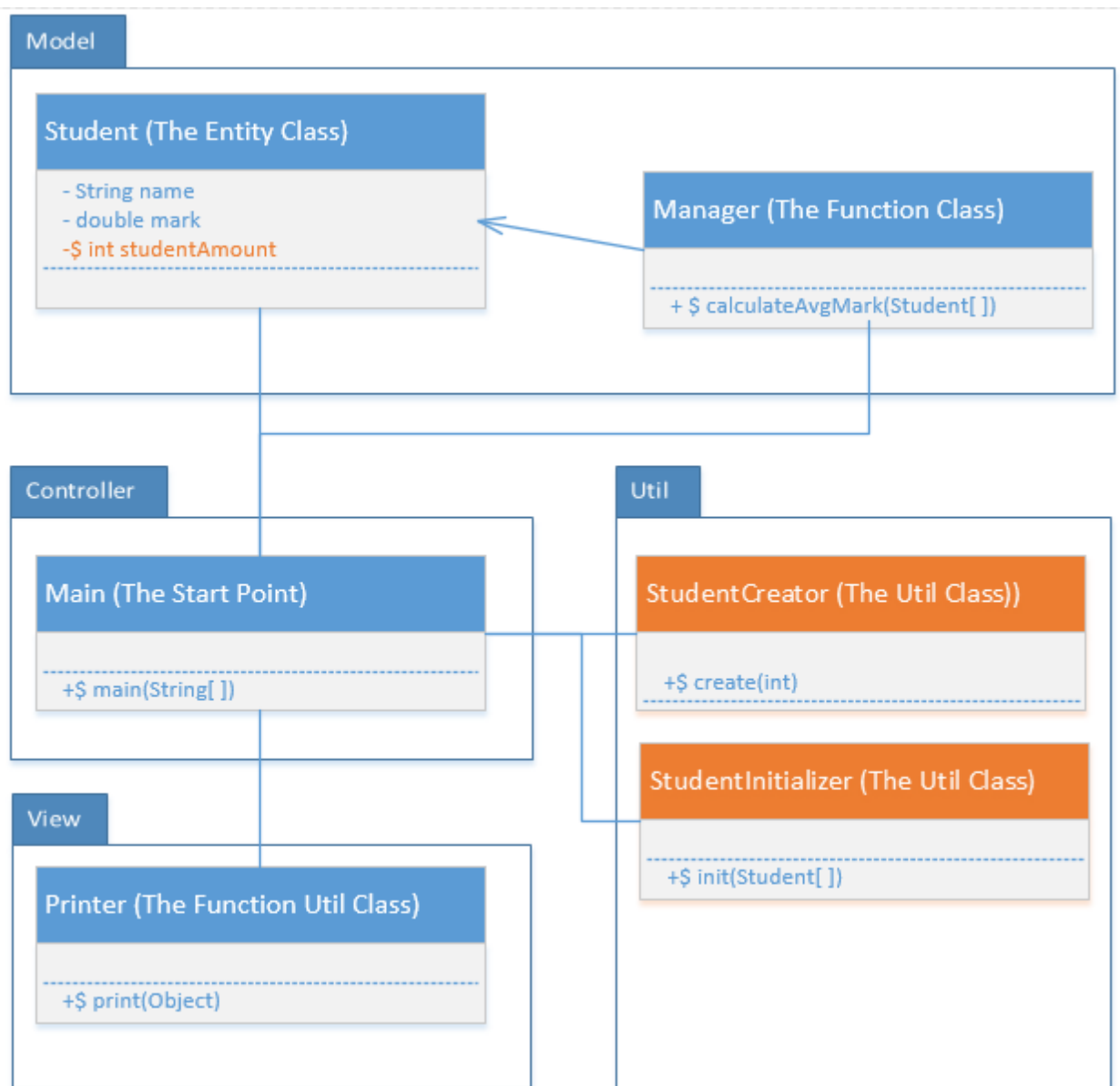


Таблица 1 – Наиболее часто используемые элементы *UML*-диаграммы

#	Shape (блок)	Description (описание)
1.		Элемент для описания пакета. Пакет логически выделяет группу классов, которые описаны в нём.
2.		Элемент для описания класса. Класс представлен в рамках, содержащих три компонента: имя класса, поля (атрибуты) класса и методы класса.
3.		Элемент для описания перечисления. Перечисление представляется аналогично классу с ключевым словом в самом вверху « <i>Enumeration</i> ».
4.		Элемент для описания интерфейса. Интерфейс представлен в рамках, содержащих два компонента: имя интерфейса с ключевым слово « <i>Interface</i> » и методы интерфейса.
5.		Аннотация (комментарий). Аннотация используется для размещения поясняющего (уточняющего) текста на диаграмме для соответствующих сущностей.
		Зависимость ( <i>Dependency</i> )
6.		Ассоциация ( <i>Association</i> )
7.		Агрегация ( <i>Aggregation</i> )
8.		Композиция ( <i>Composition</i> )
9.		Наследование ( <i>Inheritance</i> )
10.		Реализация ( <i>Implementation or Realization</i> )

## Пример модернизации проекта, который был разработан в предыдущей лабораторной работе

- 1) Дополним UML-диаграмму новыми сервисными (утилитными) сущностями: Создатель (*Creator*) и Инициализатор (*Initializer*), укажем их основные атрибуты и атрибуты, которые добавились при модернизации существующих классов, а также связи (зависимости) между новыми и основными компонентами программной системы:



- 2) Модернизируем сущность **Student**. В данную сущность добавить различные конструкторы для разнообразия инициализации состояния объекта класса:

```
package by.bntu.fitr.poisit.vikvik.university.model.entity;
```

```
public class Student {

    public static int studentAmount;

    public String name;
    public int mark;

    static {
        studentAmount = 0;
    }

    {
        studentAmount++;
    }

    public Student() {
        name = "no name";
        mark = 4;
    }

    public Student(String name, int mark) {
        this.name = name;
        this.mark = mark;
    }

    public Student(Student student) {
        name = student.name;
        mark = student.mark;
    }

    @Override
    public String toString() {
        return name + ", mark = " + mark;
    }

}
```

Имя класса должно быть именем существительным, заданным в единственном числе

Атрибут класса для хранения общего количества созданных студентов в системе

Т.е. блок инициализации вызывается каждый раз при создании объекта, то здесь и происходит приращение атрибута класса



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

- 3) Разберём более детально сущность **Student**:

```
package by.bntu.fitr.poisit.vikvik.university.model.entity;
```

```
public class Student {

    public static int studentAmount;

    public String name;
    public int mark;

}
```

Поле уровня класса для хранения общего количества созданных экземпляров данного класса

Поля уровня объекта (экземпляра класса) для хранения его состояния

```

// static initialization block (it's called only once)
static {
    studentAmount = 0;
}

// initialization block (it's called every time an object is created)
{
    studentAmount++;
}

// default constructor (constructor without arguments)
public Student() {
    name = "no name"; // default name value
    mark = 4;         // default mark value
}

// constructor with parameters
public Student(String name, int mark) {
    this.name = name;
    this.mark = mark;
}

// copy-constructor
public Student(Student student) {
    name = student.name;
    mark = student.mark;
}

@Override
public String toString() {
    return name + ", mark = " + mark;
}
}

```

Статический блок инициализации для первоначальной инициализации данных уровня всего класса

Динамический блок инициализации для инициализации состояния объекта. Вызывается при создании объекта

Конструктор по умолчанию

Конструктор с параметрами

Конструктор-копирования (разновидность конструктора с параметрами)

Переопределение метода, который обычно автоматически вызывается там, где требуется строковое представление состояния объекта

- 4) Класс бизнес-логики программы **Manager** (составная часть модели согласно паттерну MVC) в модернизации не нуждается.

```

package by.bntu.fitr.poisit.vikvik.university.model.Logic;

import by.bntu.fitr.poisit.vikvik.university.model.entity.Student;

public class Manager {
    public static double calculateAvgMark(Student[] students) {
        int total = 0;

        for (int i = 0; i < students.Length; i++) {
            total += students[i].mark;
        }
    }
}

```

```

        return total / students.Length;
    }
}

```

- 5) Также не нуждается в модернизации и класс отображения данных с использованием системной консоли **Printer** – компонент View согласно архитектурному шаблону MVC:

```
package by.bntu.fitr.poisit.vikvik.university.view;
```

```
public class Printer {
    public static void print(Object msg) {
        System.out.print(msg);
    }
}

```

Вывод данных с использованием системной консоли

- 6) Теперь опишем сервисный (утилитный) класс **StudentCreator** для создания массива студентов. Данный класс содержит статический метод, который на вход принимает целое число, которое обозначает необходимо для создания количество объектов-студентов.

```
package by.bntu.fitr.poisit.vikvik.university.util;
```

```
import by.bntu.fitr.poisit.vikvik.university.model.entity.Student;
```

```
public class StudentCreator {
    public static Student[] create(int size) {
        Student[] students = new Student[size];

        for (int i = 0; i < students.Length; i++) {
            students[i] = new Student();
        }

        return students;
    }
}

```

Создание массива

Создание шаблона объектов-студентов с помощью конструктора по умолчанию и заполнение соответствующего мас-

- 7) Далее опишем сервисный (утилитный) класс **StudentInitializer** для инициализации массива студентов. Данный класс содержит статический метод, который на вход принимает массив типа Student и производит его инициализацию или переинициализацию:

```

package by.bntu.fitr.poisit.vikvik.university.util;

import java.util.Random;

import by.bntu.fitr.poisit.vikvik.university.model.entity.Student;

public class StudentInitializer {

    public static final String[] names = {
        "Alex", "Anna", "Alexey", "Andrey", "Victor", "Peter",
        "Nikita", "Pavel", "Dima", "Denis", "Sergey", "Michael",
        "Vlad", "Vladimir", "Kristina", "Olya", "Nastya", "Igor" };

    public static final int MAX_MARK = 10;
    public static final int MIN_MARK = 0;

    public static void init(Student[] students) {
        Random random = new Random();

        for (int i = 0; i < students.length; i++) {
            students[i].name = names[random.nextInt(names.length)];
            students[i].mark = random.nextInt(MAX_MARK - MIN_MARK + 1)
                                + MIN_MARK;
        }
    }
}

```

Константный массив имён

Диапазон оценок

Инициализация (переинициализация) состояния объекта случайными («рандомными») значениями

- 8) На заключительном этапе соберём из разработанных компонентов (классов) готовую программу. Для этого перепишем класс **Main**, который выполняет роль контроллера согласно архитектурному шаблону MVC. В нём будет описан стартовый статический метод **main(...)**:

```

package by.bntu.fitr.poisit.vikvik.university.controller;

import java.util.Arrays;

import by.bntu.fitr.poisit.vikvik.university.model.entity.Student;
import by.bntu.fitr.poisit.vikvik.university.model.Logic.Manager;
import by.bntu.fitr.poisit.vikvik.university.util.StudentCreator;
import by.bntu.fitr.poisit.vikvik.university.util.StudentInitializer;
import by.bntu.fitr.poisit.vikvik.university.view.Printer;

public class Main {

    public static final int STUDENTS_NUMBER = 10;

    public static void main(String[] args) {

```

Количество тестируемых студентов

Создание объекта массива

```
Student[] group = StudentCreator.create(STUDENTS_NUMBER);
```

Инициализация объектов-студентов в созданном массиве

```
StudentInitializer.init(group);
```

```
double avgGroupMark = Manager.calculateAvgMark(group);
```

```
Printer.print(Arrays.toString(group));
```

```
Printer.print("\nAvg group mark = " + avgGroupMark);
```

```
}
}
```

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

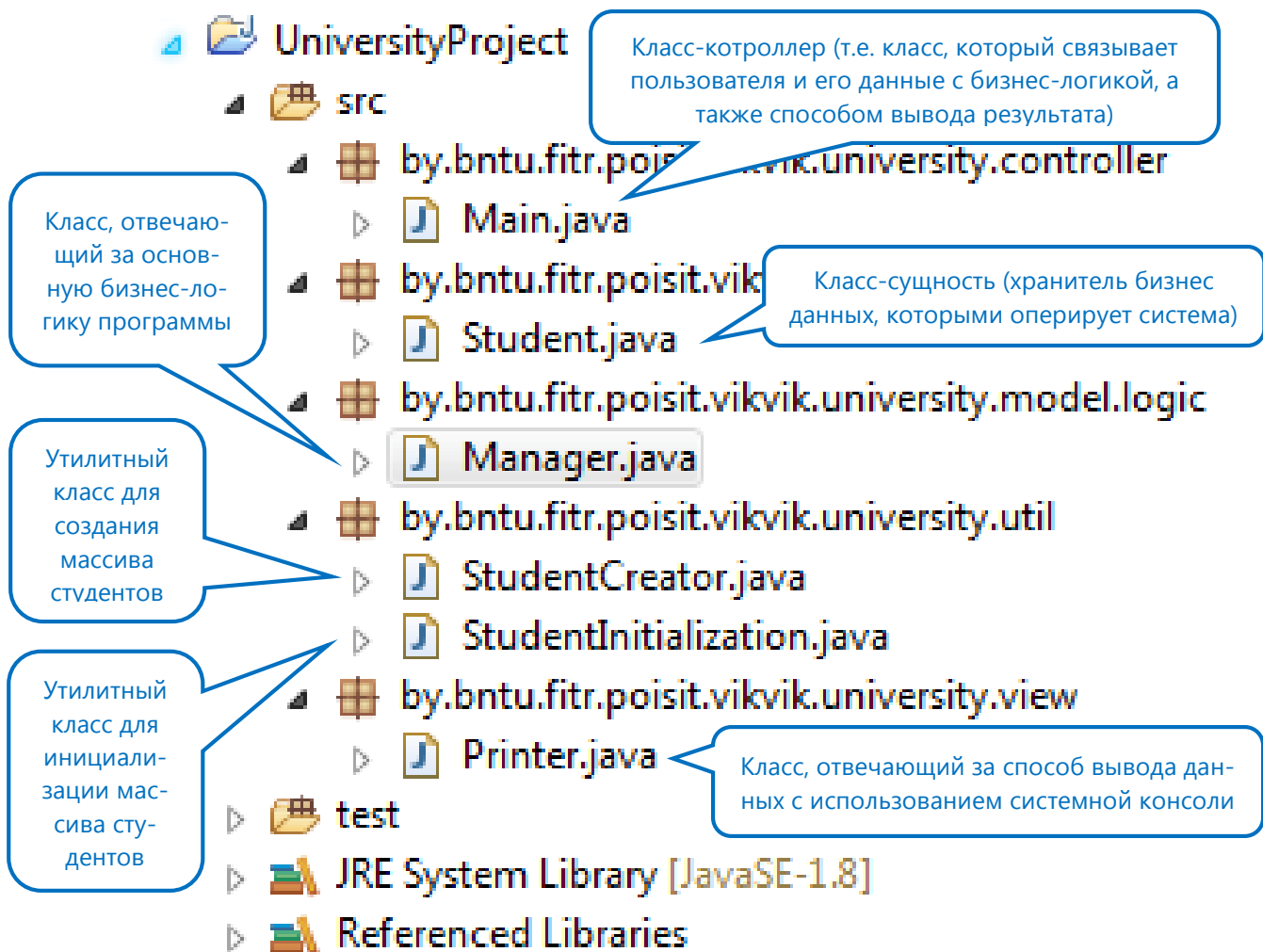


Рисунок 2 – Архитектура разработанного приложения

10) Для демонстрации работы программы перекомпилируем разработанный проект и запустим стартовый класс **Main** на выполнение:



Console

```
[Alexey, mark = 9, Anna, mark = 9, Pavel, mark = 8,  
Victor, mark = 4, Alexey, mark = 5, Vlad, mark = 9,  
Michael, mark = 10, Sergey, mark = 4, Denis, mark = 10,  
Nastya, mark = 9]
```

```
Avg group mark = 7.0
```

Рисунок 3 – Результат работы программы

## Как можно улучшить вышеописанный код?

У вышеизложенного варианта реализации задания есть ряд серьёзных ошибок, которые могут привести к краху всей программы или к неверному результату. Попробуйте найти и устранить данные ошибки.