



Laboratory Work #22

# Java Object-Oriented Programming. Object Initialization. Class Static Components



**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 #22

## ООП в Java. Инициализация состояния объекта. Статические компоненты класса

### Цель работы

Научиться грамотно использовать соответствующие средства, предоставляемые языком Java, для первоначальной инициализации состояния объекта, а также изучить истинное предназначение статических компонентов класса в языке Java.

### Требования

- 1) Необходимо скорректировать UML-диаграмму взаимодействия классов и объектов программной системы с учётом вносимых дополнений и изменений.
- 2) 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 дополнения:

- для грамотной инициализации состояния объектов соответствующей предметной области добавить всевозможные средства инициализации, которые предоставляет язык Java (блоки инициализации, конструктор по умолчанию, конструкторы с параметрами, конструктор-копирования и т.д.);
- проанализировав соответствующую предметную область добавить в проект статические компоненты класса и возможность их первоначальной инициализации с помощью средств, который предоставляет язык Java.

Дополнительно необходимо проанализировать стадии и способы инициализации как состояния объектов, так и состояния соответствующих объектов-классов (объектов класса *Class*), а также их очередность вызова JVM. Привести анализ результатов и соответствующие выводы в отчёте.

## Что нужно запомнить (краткие тезисы)

1. **Инстанцирование** – процесс создания и инициализации объекта (экземпляра класса).
2. **Каждый класс** может иметь специальные методы, которые автоматически вызываются после создания и(или) перед уничтожением экземпляров класса:
  - **конструктор (*constructor*)** – служит для первоначальной инициализации состояния объекта и автоматически вызывается сразу же после создания объекта в памяти;
  - **деструктор (*destructor*)** – служит в основном для освобождения всех ресурсов, которые были выделены для работы текущего объекта, и автоматически вызывается перед полным удалением объекта из памяти сборщиком мусора (*garbage collector, GC*);
3. В ООП конструктор – это специальный метод класса, обеспечивающий создание и инициализацию экземпляра класса.
4. В ООП деструктор – это специальный метод класса, обеспечивающий уничтожение экземпляра, относящегося к определённому классу.
5. В Java конструктор – это метод, который не имеет имени, а в качестве возвращаемого значения указывается всегда тип текущего класса.
6. В языке Java в классе можно описать несколько типов (видов) конструкторов:
  - **конструктор по умолчанию**, или ***constructor with no arguments***, или ***default-constructor*** (конструктор, который не принимает извне ни одного аргумента (значения) для первоначальной пользовательской инициализации состояния объекта);
  - **конструкторы с параметрами** (конструкторы, которые могут принимать различное количество и типы аргументов для первоначальной инициализации состояния объекта);
  - **конструктор-копирования** (конструктор, который используется для создания эквивалентного по состоянию объекта текущего класса и в качестве единственного параметра принимает ссылку на объект, у которого должно быть скопировано состояние).
7. В языке Java внутри класса невозможно явно объявить деструктор. Роль деструктора в языке выполняет специальный метод ***finalize()***, который наследуется всеми классами от базового класса *Object*. Его можно переопределить.

8. Дополнительными средствами инициализации состояния объекта в языке Java являются **блоки инициализации**, которые описываются внутри класса и похожи на описание методов, но, в отличие от методов, имеют в своём составе только тело, описанное в фигурных скобках.
9. Статические блоки инициализации служат для инициализации статического содержимого класса. Они вызываются в порядке их объявления в классе и только один раз во время загрузки соответствующего класса в память.
10. Динамические блоки инициализации служат для инициализации содержимого создаваемого объекта класса. Они вызываются в порядке их объявления в классе каждый раз, когда создаётся экземпляр данного класса.
11. Из всех инициализаторов, которые есть в языке Java, конструкторы выполняются в самую последнюю очередь.

# 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.

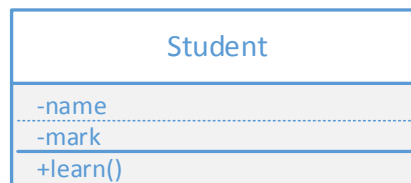
Словарь UML включает три вида строительных блоков: **диаграммы, сущности** и **связи**. **Сущности** – это абстракции, которые являются основными элементами модели, **связи** соединяют их между собой, а **диаграммы** группируют представляющие интерес наборы сущностей.

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.

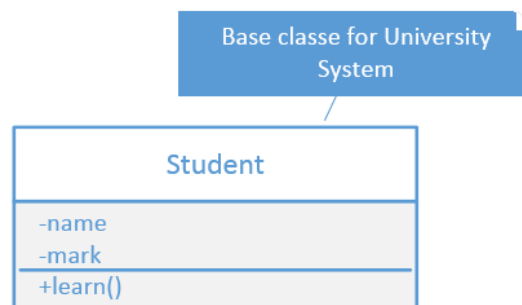
**Структурные сущности** – это «имена существительные» в модели UML. В основном, статические части модели, представляющие либо концептуальные, либо физические элементы. Основным видом структурной сущности в диаграммах классов является класс. Пример класса Студент (*Student*) с полями и методами:



**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.

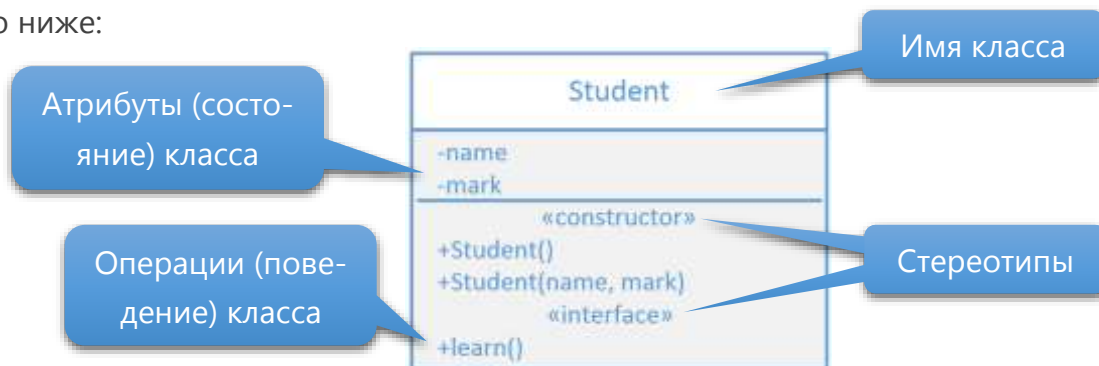
**Абстрактные методы** класса обозначаются **курсивным** шрифтом.

**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.

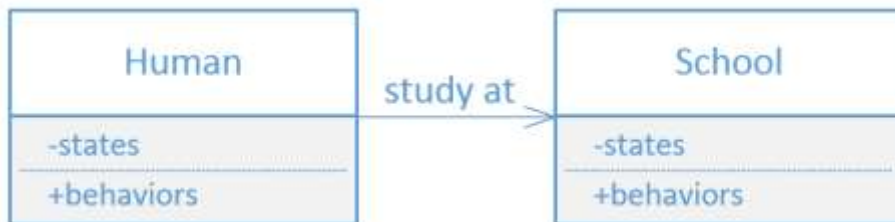
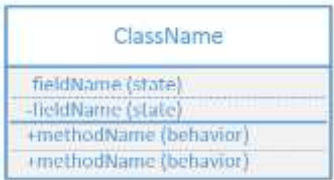


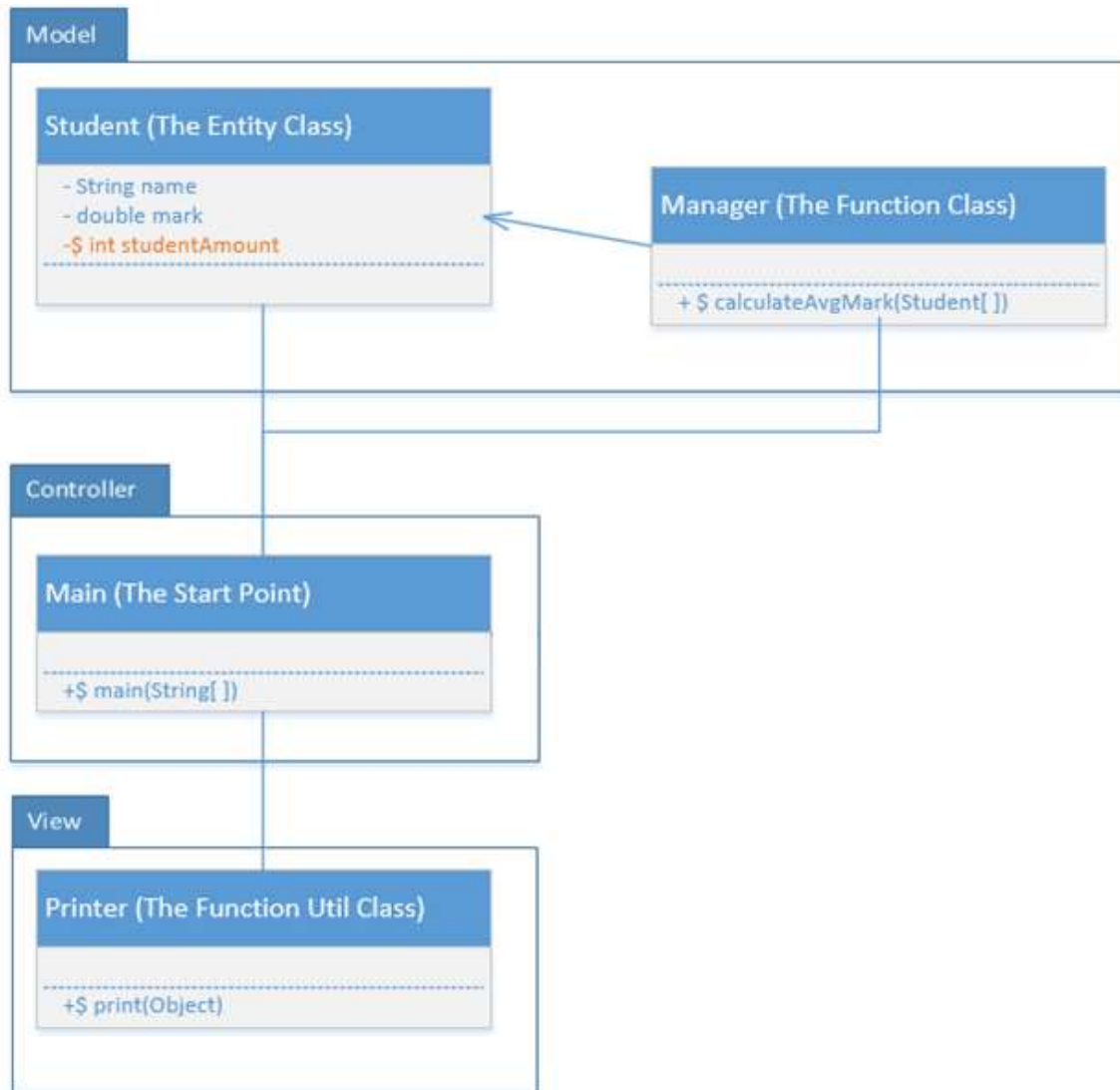


Таблица 1 – Наиболее часто используемые элементы UML-диаграммы

#	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 верху « <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) Обновим UML-диаграмму:



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

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

```
public class Student {
```

Имя класса должно быть именем существительным, заданным в единственном числе

```

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;
}
}

```

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;
}

```

Статический блок инициализации для первоначальной инициализации данных уровня всего класса

```
// 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;
}
```

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) 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;
```

Импорт стандартного утилитного класса **Arrays** для работы с массивами

Секция импорта для обращения в текущем коде к типам по простому имени

```
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.view.Printer;
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        Student[] group = {
            new Student("Alex", 10),
            new Student("Nikita", 9),
            new Student("Nastya", 8),
            new Student("Max", 7),
            new Student("Peter", 6),
            new Student("Ivan", 5),
            new Student("Vlad", 4)};
```

Создание объекта-массива и его элементов

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

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

```
    }
}
```

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

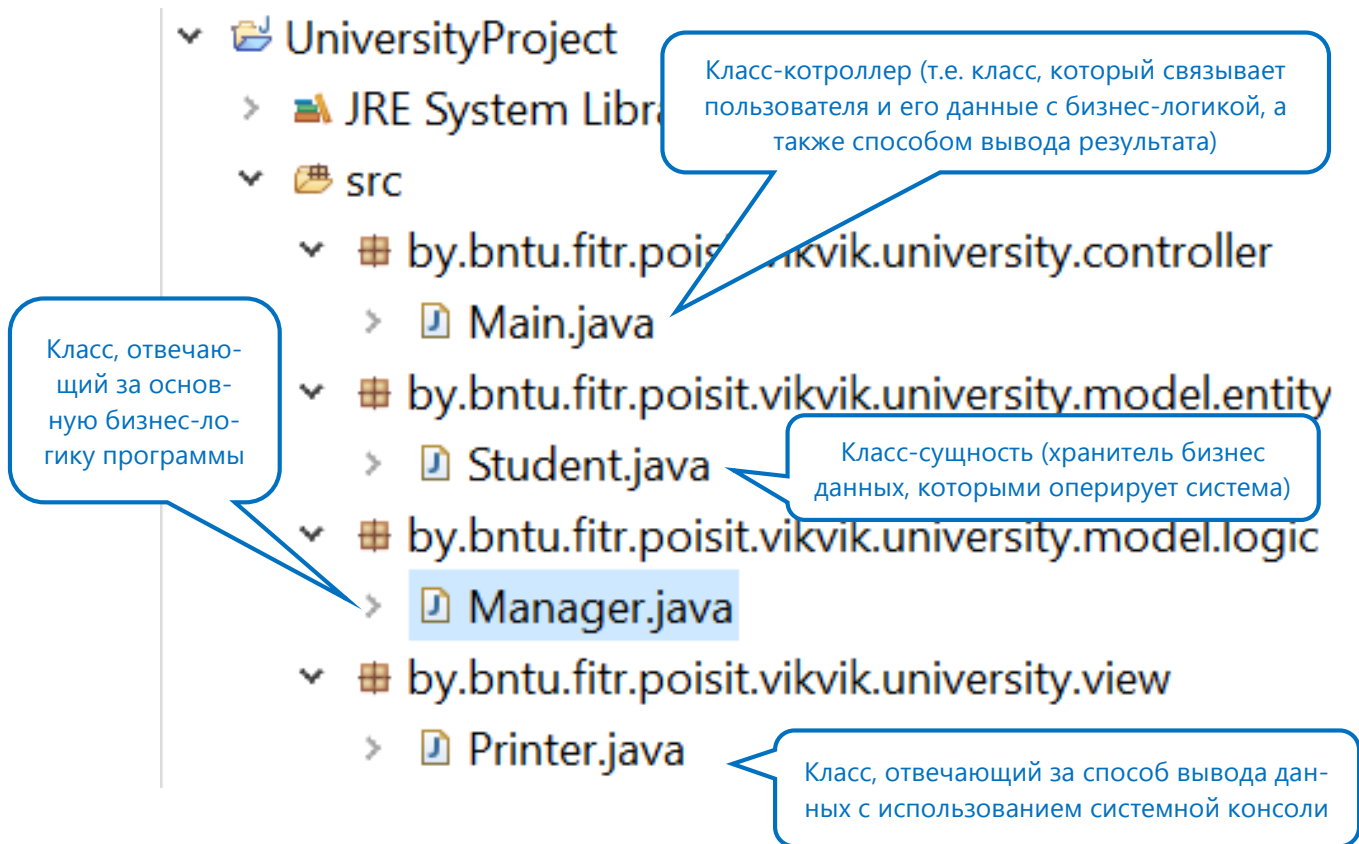


Рисунок 2 – Архитектура разработанного приложения

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

```
Console
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-11\bin\javaw.exe
List of students:
Alex, mark = 10
Nikita, mark = 9
Nastya, mark = 8
Max, mark = 7
Peter, mark = 6
Ivan, mark = 5
Vlad, mark = 4

Avg group mark = 7.0
```

Рисунок 3 – Результат работы программы

## Как 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.