



Laboratory Work #10

Basics Object-Oriented Programming in Java. Abstraction. Object Initialization



LEARN. GROW. SUCCEED.

© 2018-2019. Department: <Software of Computer Technology & Computer-Aided Systems>
Faculty of Information Technology and Robotics
Belarusian National Technical University
by Viktor Ivanchenko / Minsk

ЛАБОРАТОРНАЯ РАБОТА #10

Основы Объектно-Ориентированного Программирования в Java. Абстракция. Инициализация объектов

Цель работы

Научится решать проблемы с помощью объектно-ориентированного подхода (выделять из предметной (доменной) области основные сущности (абстракции); проектировать и реализовывать на базе выделенной абстракции классы; создавать на базе классов объекты и применять к ним различные способы инициализации).

Требования

- 1) Необходимо спроектировать и реализовать UML-диаграмму взаимодействия классов и объектов разрабатываемой программной системы с отображением всех связей (отношений) между классами и объектами.
- 2) При проектировании и разработке системы необходимо полностью использовать своё объектно-ориентированное воображение и по максимум использовать возможности, которые предоставляет язык программирования Java для реализации ООП-методологии.
- 3) Основные классы системы должны быть самодостаточными, т.е. не зависеть, к примеру, от консоли! Любые типы отношений между классами должны применяться обосновано и лишь тогда, когда это имеет смысл.
- 4) На базе спроектированной программной системы реализуйте простейшее интерактивное консольное приложение. Используйте при реализации архитектурный шаблона проектирования **Model-View-Controller, MVC**.
- 5) Создаваемые классы необходимо грамотно разложить по соответствующим пакетам, которые должны иметь «адекватные» названия и быть вложены в указанные стартовые пакеты: **by.bntu.fitr.povt.nameofteam.javalabs.lab10**.

- 6) Попробуйте реализовать все средства инициализации при создании соответствующих объектов программной системы.
- 7) При выполнении задания необходимо по максимуму пытаться разрабатывать универсальный, масштабируемый и легко поддерживаемый и читаемый код.
- 8) 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.
- 9) В соответствующих компонентах бизнес-логики необходимо предусмотреть «защиту от дурака».
- 10) Программа должна обязательно быть снабжена комментариями, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и её название, версию программы, ФИО разработчиков, название бригады (если есть), номер группы и дату разработки. Исходный текст классов и демонстрационной программы рекомендуется также снабжать поясняющими краткими комментариями.
- 11) Программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом для взаимодействия с пользователем.
- 12) Интерфейс программы и комментарии должны быть на английском языке.
- 13) При проверке работоспособности приложения необходимо проверить все тестовые случаи.
- 14) При выполнении задания не рекомендуется использовать интегрированные средства разработки (*Integrated Development Environment, IDE*). Лучше задействовать любой текстовый редактор и основные компоненты Java (компилятор – **javac**, утилиту для запуска JVM – **java**, утилиту для создания JAR-файлов – **jar**).
- 15) При разработке программ придерживайтесь соглашений по написанию кода на Java (**Java Code-Convention**) !!!

Основное задание



Необходимо решить задачу с использованием методологии ООП. Для чего необходимо подобрать самостоятельно соответствующую проблемную (предметную или доменную) область, которая базируется на объектах реального мира. Спроектировать классы (пользовательские типы данных) в Java для программного представления данных объектов и основной логики системы. Программа должна решать, как минимум, два полезных действия и иметь следующие вещи:

- не менее 3 разнообразных классов предметной области;
- не менее 5 атрибутов (характеристик) в каждом соответствующем классе-сущности;
- не менее 3 методов (поведения) в соответствующих функциональных классах;
- хранить глобальные характеристики системы или характеристики уровня отдельных классов;
- всевозможные средства инициализации (блоки инициализации, конструкторы по умолчанию, конструкторы с параметрами, конструкторы-копирования и т.д.) для объектов классов-сущностей.

Написать программу для создания объектов спроектированной системы и демонстрации взаимодействия между ними.

Дополнительно необходимо проанализировать стадии и способы инициализации состояния объекта и их очередность вызова JVM при создании соответствующих объектов. Отобразить результат и выводы в отчёте.

Дополнительное задание

Необходимо переработать программу из дополнительного задания предыдущей лабораторной работы таким образом, чтобы пользователь загадывал число, а компьютер его отгадывал, используя более эффективный алгоритм. Для реализации «интеллекта» компьютера разработайте минимум два подхода (на базе бинарного поиска (рациональный подход) и с использованием генератора случайных чисел (азартный подход)).

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



Что нужно запомнить (краткие тезисы)

1. **ООП** использует в качестве базовых элементов объекты, а не алгоритмы.
2. **Объект** – это понятие, абстракция или любой предмет с чётко очерченными границами, имеющий смысл в контексте рассматриваемой прикладной проблемы. Введение объекта преследует две цели: *понимание прикладной задачи (проблемы)* и *введение основы для реализации на компьютере*.
3. **Главная идея ООП** – всё состоит из объектов! Программа, написанная с использованием ООП, состоит из множества объектов, и все эти объекты взаимодействуют между собой посредством посылке (передачи) сообщений друг другу. ООП и реальный мир не могут существовать отдельно!
4. **Программные объекты** могут представлять собой объекты реального мира или быть полностью абстрактными объектами, которые могут существовать только в рамках программы. **Гради Буч** (создатель унифицированного языка моделирования UML) даёт следующее определение объекта: «**Объект – это мыслимая или реальная сущность, обладающая характерным поведением, отличительными характеристиками и являющая важной в предметной области**».
5. **Каждый объект** имеет состояние, обладает некоторым хорошо определённым поведением и уникальной идентичностью.
6. **Состояние (state)** (синонимы: параметры, аспекты, характеристики, свойства, атрибуты, ...) – совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой конкретный момент времени состояние объекта включает в себя перечень параметров объекта и текущее значение этих параметров. Состояние объекта в Python реализуется с помощью описания полей класса.
7. **Поведение (behavior)** – действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта. Поведение объекта в Python реализуется с помощью описания методов класса.
8. **Уникальность (identity)** – эта природа объекта; то, что отличает один объект от других. В машинном представлении уникальность объекта – это адрес размещения объекта в памяти. Следовательно, уникальность объекта состоит в том, что всегда можно определить, указывают две ссылки на один и тот же объект, или на разные объекты.

9. Чтобы создать программный объект или группу объектов необходимо вначале описать где-то его(их) характеристики и шаблон поведения. Для этих целей в ООП существуют классы (*classes*). Формально, **класс** – это шаблон поведения объектов определённого типа с определёнными параметрами, которые описывают состояние объекта.
10. **Все экземпляры** (объекты) одного и того же класса имеют один и тот же набор характеристик (значение которых может быть различным у разных объектов) и общее поведение (все объекты одинаково реагируют на одинаковые сообщения).
11. В упрощённом виде, класс – это кусок кода, у которого есть имя. Чтобы воспользоваться данным кодом, нужно создать объект (**экземпляр класса**).
12. **Инстанцирование** – процесс создания и инициализации объекта (экземпляра класса).
13. **Каждый класс** может иметь также специальные методы, которые автоматически вызываются при создании и(или) уничтожении объектов класса:
 - **конструктор (*constructor*)** – служит для первоначальной инициализации состояния объекта и выполняется сразу же после создания объекта в памяти;
 - **деструктор (*destructor*)** – служит для освобождения всех ресурсов, которые были выделены для объекта, и выполняется перед полным удалением объекта из памяти сборщиком мусора (*garbage collector, GC*);
14. В ООП конструктор – это специальный метод класса, обеспечивающий создание и инициализацию объекта данного класса
15. В ООП деструктор – это специальный метод, обеспечивающий уничтожение объекта, относящегося к определённому классу.
16. В Java конструктор – это метод, не имеющий при декларации возвращаемого типа значения и имя которого совпадает с именем класса.
17. В Java 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 созданного объекта);

- **конструкторы с параметрами** (конструкторы, которые могут принимать различное количество и типы аргументов для первоначальной инициализации состояния созданного объекта);
- **конструктор-копирования** (конструктор, который в качестве единственного параметра принимает ссылку на объект аналогичного класса, объект которого был создан).

18. В Java внутри класса невозможно явно объявить деструктор.

19. Роль деструктора в Java заменяет специальный метод ***finalize()***, который наследуется всеми классами от базового класса *Object* и который нужно переопределить в целевом классе, если в этом есть необходимость.

20. Дополнительными средствами инициализации состояния созданного объекта в 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.

21. 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.

22. Динамические блоки инициализации служат для инициализации содержимого создаваемого объекта класса. Они вызываются в порядке их объявления в классе каждый раз, когда создаётся объект данного класса.

23. Тело конструктора выполняется в самую последнюю очередь.

24. ООП помогает:

- ✓ уменьшить **сложность** программного обеспечения (ПО);
- ✓ увеличить **производительность** труда программистов и **скорость** разработки ПО;
- ✓ *повысить* **надёжность** ПО;
- ✓ обеспечить возможность лёгкой **модификации** отдельных компонентов ПО без изменения остальных его частей;
- ✓ обеспечить возможность **повторного использования** отдельных компонентов ПО.

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.

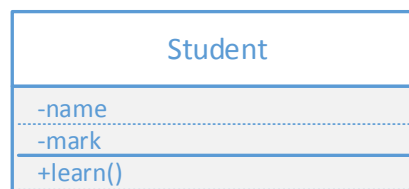
Словарь UML включает три вида строительных блоков: **диаграммы, сущности** и **связи**. **Сущности** – это абстракции, которые являются основными элементами модели, **связи** соединяют их между собой, а **диаграммы** группируют представляющие интерес наборы сущностей.

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.

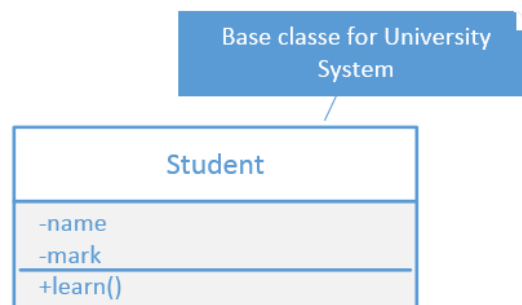
Структурные сущности – это «имена существительные» в модели UML. В основном, статические части модели, представляющие либо концептуальные, либо физические элементы. Основным видом структурной сущности в диаграммах классов является класс. Пример класса Студент (*Student*) с полями и методами:



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).

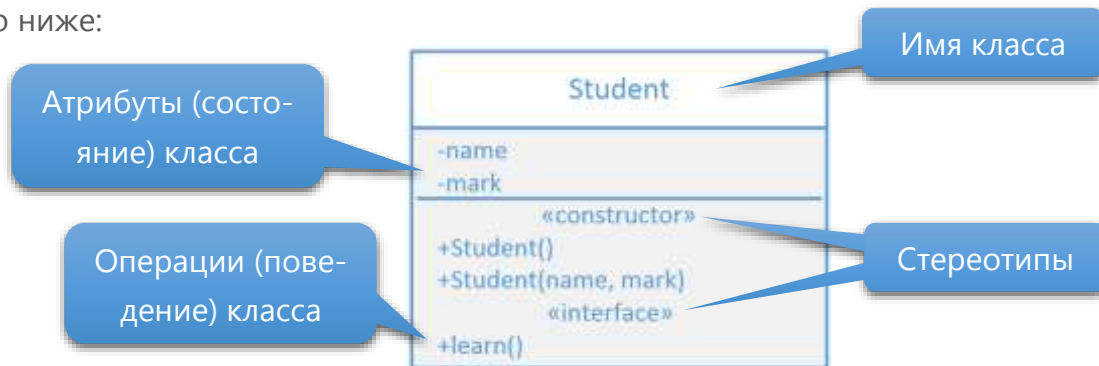
Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения.

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.

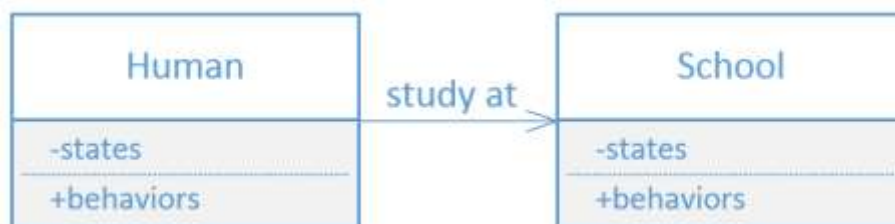
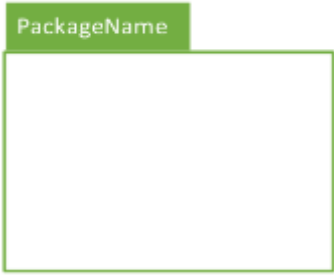
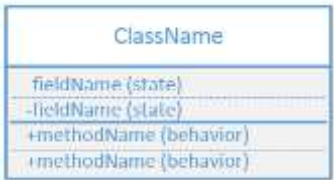
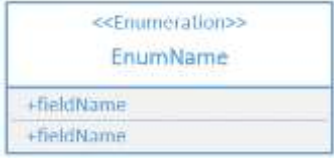
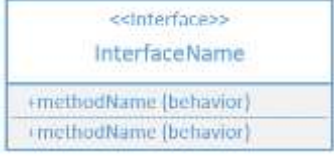









Таблица 1 – Наиболее часто используемые элементы UML-диаграммы

#	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>)

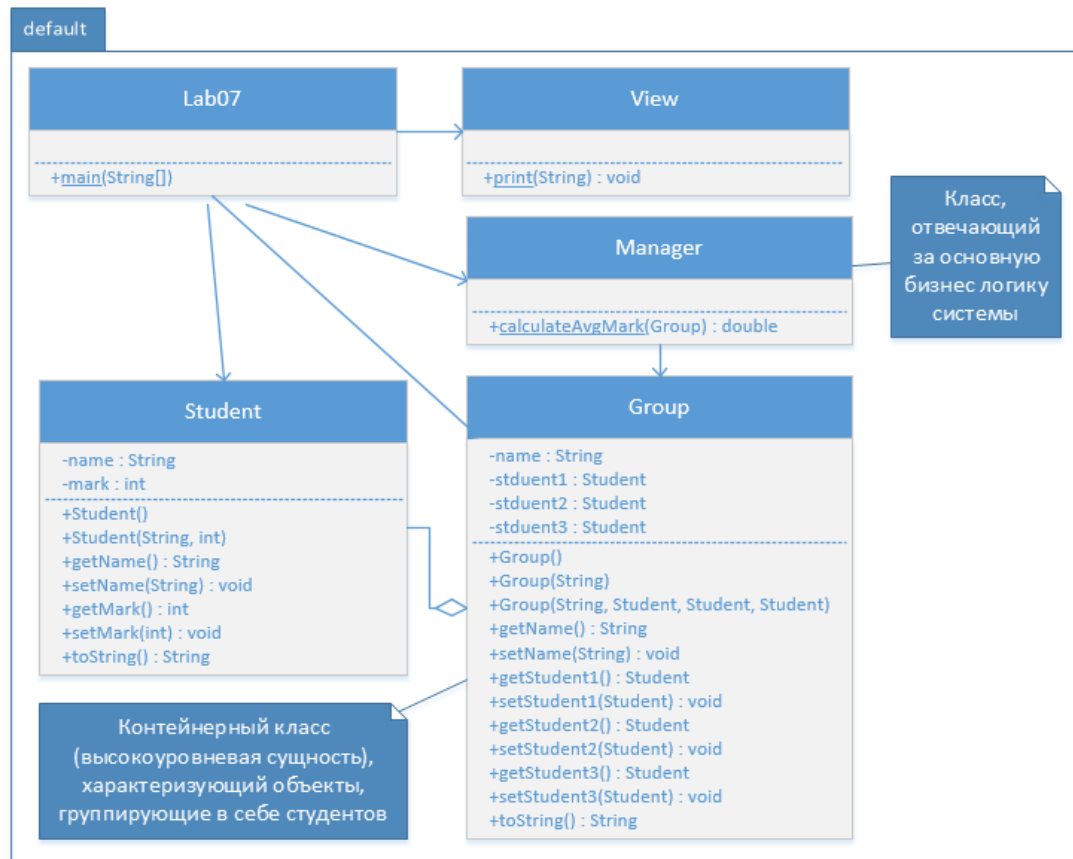
Пример выполнения практического задания с использованием Java классов и объектов, а также архитектурного шаблона проектирования MVC

Задание

В данный момент в группе три студента: Иванов, Петров и Сидоров. Спроектируйте и реализуйте программную систему, которая бы хранила информацию о данных студентах (к примеру, имя и оценку) и подсчитывала среднюю успеваемость студентов группы. Разрешается данные студентов и группы задавать «хардкодом» в тестовом классе.

Решение

- 1) Выделим основные сущности нашей предметной области (класс Студент (*Student*), класс Группа (*Group*), класс бизнес логики Заведующий кафедрой (*Manager*), класс для вывода данных *View* и класс-контроллер *Lab10*) и спроектируем *UML*-диаграмму взаимодействия классов и объектов приложения:



- 2) Опишем сущность *Student*. Данная сущность ответственна за хранение соответствующей информации о студентах. Класс данной сущности содержит в себе Ф.И.О. студента (*name*), оценку (*mark*), соответствующие блоки инициализации и конструкторы для инициализации начального состояния сущности, а также переопределённый метод *toString()*:

```
package by.bntu.fitr.povt.vikvik.javalabs.lab10.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 + ")";
    }

}
```

Имя класса должно быть именем существительным, заданным в единственном числе



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

- 3) Разберём более детально сущность *Student*

```
package by.bntu.fitr.povt.vikvik.javalabs.lab10.model.entity;
```

```
public class Student {

    public static int studentAmount;
```

Поле уровня класса для хранения общего количества созданных объектов данного класса

```

public String name;
public int mark;

// 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 + ")";
}
    
```

Поля уровня объектов (экземпляров класса) для хранения их состояния

Статический блок инициализации для первоначальной инициализации данных уровня всего класса

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

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) Далее опишем сущность *Group*. Данная сущность состоит из названия группы (*name*), трёх студентов (соответствующие поля *student1*, *student2* и *student3*), соответствующих блоков инициализации и конструкторов для инициализации высокоуровневой сущности, а также переопределённый метод *toString()* для вывода содержимого сущности:


```

package by.bntu.fitr.povt.vikvik.javalabs.Lab10.model.entity;

public class Group {
    public static final int DEFAULT_STUDENT_AMOUNT = 3;

    public String name;
    public Student student1;
    public Student student2;
    public Student student3;

    // default constructor (constructor without parameters)
    public Group() {
    }

    // constructor with parameters
    public Group(String name) {
        this.name = name;
    }

    // constructor with parameters
    public Group(String name, Student st1, Student st2, Student st3) {
        this.name = name;
        student1 = st1;
        student2 = st2;
        student3 = st3;
    }

    // copy-constructor (type of constructor with parameters)
    public Group(Group group) {
        name = group.name;
        student1 = group.student1;
        student2 = group.student2;
        student3 = group.student3;
    }

    @Override
    public String toString() {
        return "Group " + name + ":\n" + student1 + "\n" + student2 + "\n" +
        student3;
    }
}
    
```

Имя класса должно быть именем существительным, заданным в единственном числе

Константное поле уровня всего класса

Описание состояния сущности Группа

- 5) Теперь составим блок-схему и реализуем бизнес-логику программы (составная часть модели согласно паттерну MVC) в соответствующем статическом методе класса *Manager* – *calculateAvgMark(Group group)*. Данный метод предназначен для вычисления средней успеваемости группы студентов. Он принимает на вход один параметр – ссылочную переменную, которая ссылается на

объект группы, и возвращает среднеарифметическое трёх оценок соответствующих студентов группы:

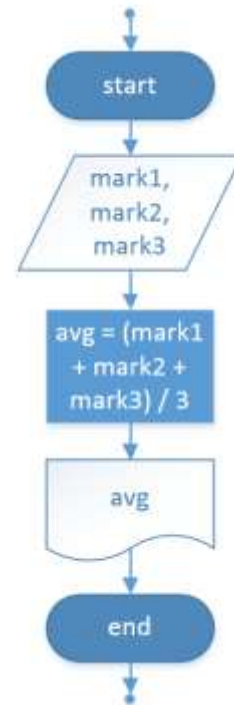
```
package by.bntu.fitr.povt.vikvik.javababs.Lab10.model.Logic;

import by.bntu.fitr.povt.vikvik.javababs.Lab10.model.entity.*;

public class Manager {
    public static double calculateAvgMark(Group group) {
        int total = group.student1.mark
            + group.student2.mark
            + group.student3.mark;

        return total / Group.DEFAULT_STUDENT_AMOUNT;
    }
}
```

Рисунок 1 – Блок-схема алгоритма вычисления средней успеваемости конкретной группы, состоящей из трёх студентов



- 6) Один из последних классов, который необходимо реализовать, это класс отображения данных с использованием системной консоли *Printer* – компонент *View* согласно архитектурному шаблону *MVC*:

```
package by.bntu.fitr.povt.vikvik.javababs.Lab10.view;

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

Вывод данных с использованием системной консоли

- 7) На заключительном этапе соберём из разработанных компонентов (классов) готовую программу. Для этого опишем класс *Lab10*, который будет выполнять роль контроллера согласно архитектурному шаблону *MVC*. В нём будет описан стартовый статический метод **main(...)**:

```
package by.bntu.fitr.povt.vikvik.javababs.Lab10.controller;

import by.bntu.fitr.povt.vikvik.javababs.Lab10.model.entity.Group;
import by.bntu.fitr.povt.vikvik.javababs.Lab10.model.entity.Student;
import by.bntu.fitr.povt.vikvik.javababs.Lab10.model.Logic.Manager;
import by.bntu.fitr.povt.vikvik.javababs.Lab10.view.Printer;
```

```

public class Lab10 {

    public static void main(String[] args) {

        Group group = new Group("POIT10701217");
        group.student1 = new Student("Ivanov", 10);
        group.student2 = new Student("Petrov", 5);
        group.student3 = new Student("Sidorov", 9);

        double avgGroupMark = Manager.calculateAvgMark(group);

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

    }
}
    
```

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

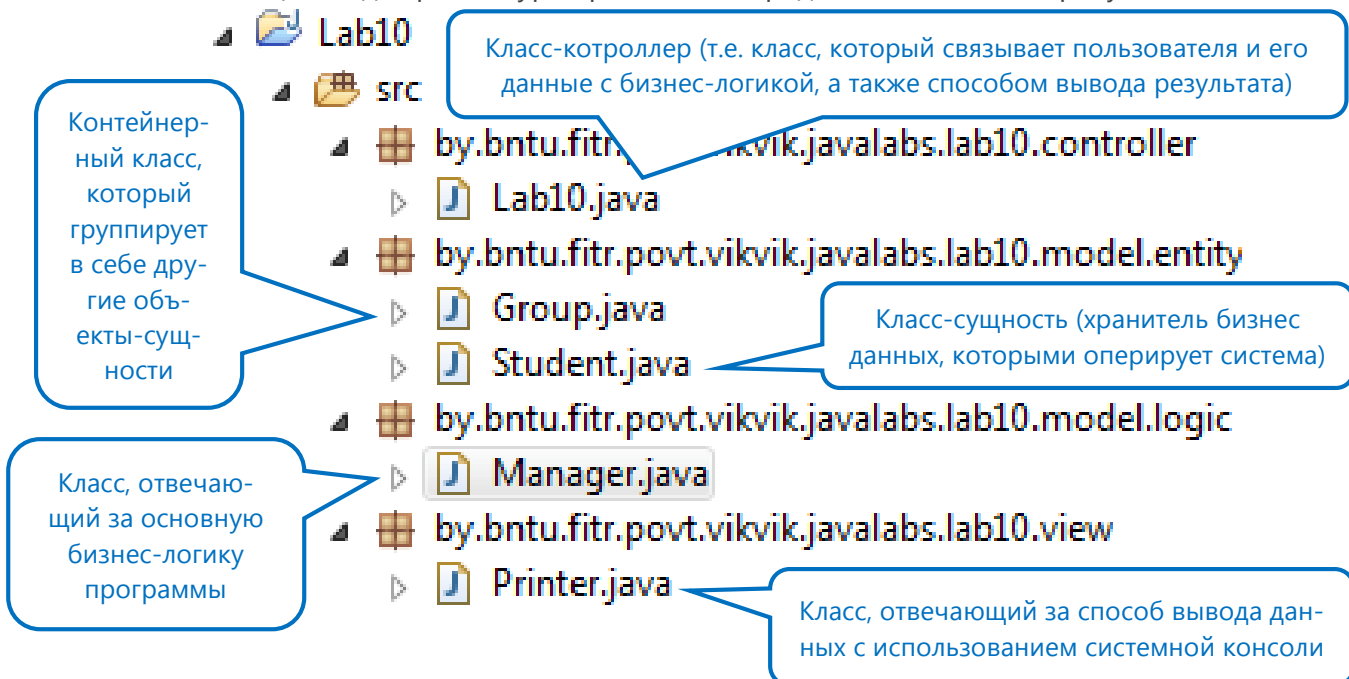


Рисунок 2 – Архитектура разработанного приложения

9) Для демонстрации работы программы перекомпилируем разработанный проект и запустим стартовый класс `Lab10` на выполнение:

```

Console
Group POIT10701217:
Ivanov (10)
Petrov (5)
Sidorov (9)
Avg group mark = 8.0
    
```

Рисунок 3 – Результат работы программы при вводе положительного числа

Как улучшить вышеописанный код?

- 1) 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.
- 2) Как следует изменить архитектуру и код решения задания, чтобы данная реализация была более универсальной (масштабируемой) и могла подсчитывать среднюю успеваемость группы с любым количеством студентов, и при этом, не надо было бы переписывать код?

7 смертных грехов программирования



Усман Шаукат, опыт в сфере веб-разработки: PHP, JavaScript, Node.js, ...

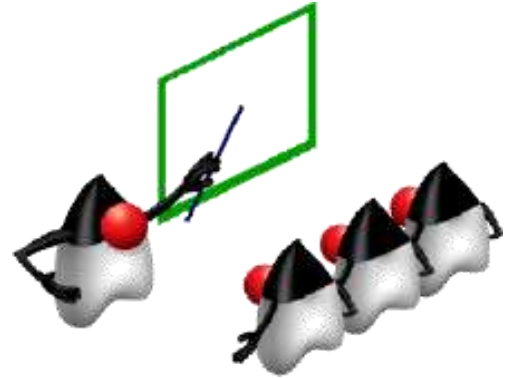
1. **Программировать, не планируя.** Самый страшный из всех грехов.
2. **Пытаться изобрести колесо.** Если есть возможность, всегда используйте алгоритмы, предложенные в книгах и научных статьях (например, алгоритмы сортировки, поиска и т.д.), а не пишите собственные.
3. Писать несистематизированные/некачественные коды и **не придерживаться стандартов программирования.**
4. Считать, что тестирование – это не ваша забота. Я вас очень прошу, пожалуйста, **тестируйте свои коды.**
5. Писать сложный код, когда с тем же успехом можно обойтись простым. **Простые коды – это элегантно.**
6. Слепое копирование-вставка с сайтов вроде stackoverflow.com без ознакомления с пояснениями и комментариями.
7. Последнее, и самое важное – **совершенствуйтесь сами и осваивайте новый инструментарий.** Никогда не бойтесь новшеств. Знакомьтесь с ними раньше всех. Это поможет вам оставаться востребованным.

Source: <https://www.kv.by/post/1053298-7-smertnyh-grehov-programmirovaniya>

how to think...

Числовые последовательности

Всем известно, что стимуляция мозга способствует росту новых мозговых клеток и новым связям между ними – что надёжно защищает будущее здоровье мозга и долгую превосходную работу памяти.



Который из представленных в нижнем ряду вариантов числовых квадратов (A, B, C или D) воспроизводит принцип первых трёх числовых квадратов в верхней последовательности?

15	17	20	12	14	17	17	19	22			
24	29	35	21	26	32	26	31	37		?	
42	50	59	39	47	56	44	52	61			
13	15	18	11	13	16	16	19	21	14	16	19
22	27	33	20	25	31	25	30	36	24	28	34
40	49	57	38	46	55	43	51	60	41	49	58
A			B			C			D		