МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра веб-технологий и компьютерного моделирования

Курсовая работа

Методические указания и рекомендации по применению классов в Free Pascal

Выполнил:  
Студент 2-ого курса 3-ей группы  
Трифонов Вадим

Минск, 2019

Оглавление

[Оглавление 2](#_Toc7596047)

[Введение 3](#_Toc7596048)

[Понятие объектно-ориентированного программирования 4](#_Toc7596049)

[Абстрактные типы данных 5](#_Toc7596050)

[Инкапсуляция 6](#_Toc7596051)

[Наследование 7](#_Toc7596052)

[Конструкторы и деструкторы 8](#_Toc7596053)

[Виртуальные и статические методы 9](#_Toc7596054)

[Полиморфизм 10](#_Toc7596055)

[Литература 11](#_Toc7596056)

Введение

Данное учебно-методическое пособие предназначено для ознакомления и введению в тему “Объектно-Ориентированного Программирования на языке Pascal”.  
Основная цель – максимально понятно и доступно показать каким образом реализуется идеология ООП(Объектно-Ориентированное Программирование) на языке Pascal, сформулировать основные принципы использования. В данном методическом пособии будет описываться основной теоретический материал для каждой из тем и подкрепляться различными задачами с их дальнейшим решением. Это поможет лучше освоить материал и попробовать его самому для лучшего понимания.   
В методичке не изложен материал для изучения азов языка Pascal, предполагается, что вы уже знакомы с основами языка.  
  
Все примеры в учебном пособии были написаны в среде: Free Pascal версии 1.0.12

Понятие объектно-ориентированного программирования

Объектно-Ориентированное Программирование (ООП) – идеология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса.

**Классы и объекты.**В ООП существует два ключевых понятия: **Класс** и **Объект.**

* **Класс** – это абстрактный тип данных (будет рассмотрено позже). Позволяющий описать некоторые характеристики и возможные действия с его экземпляром.
* **Объект** – конкретный представитель класса.

Объекты в ООП представляется в виде некоторых блоков кода, которые относятся к определённому заранее классу. Данные, которые содержит объект, принято называть *атрибутами объекта*. Они позволяют описать состояние объекта и его характеристики.  
Если же нам необходимо взаимодействовать с атрибутами некоторого объекта или же описать его поведение, то необходимо задать процедуры и функции для этого объекта. В ООП функции и процедуры объектов называют *методами*. Но задавать мы их будем не для каждого объекта по отдельности, а для класса, которому принадлежит этот объект. Вызов метода происходит путём обращения к объекту и через точку задается имя вызываемого метода (ИмяОбъекта.ИмяМетода), это происходит в связи с тем, что вызванный метод работает с данными того объекта, относительно которого она был вызван (позже будет приведён пример). Каждый метод имеет доступ к атрибутам объекта.  
Класс является некоторой базой, обладающей всей информацией необходимой для создания объекта: общими данными, характерными для каждого объекта данного класса; описанием поведения объектов (методов)

Перейдём к реализации в FreePascal. Для корректной работы с ООП необходимо подключить директиву компилятора {$MODE OBJFPC}.Для начала необходимо описать наш первый класс в секции type, делается это путём использования ключевого слова Class (Рисунок 1):

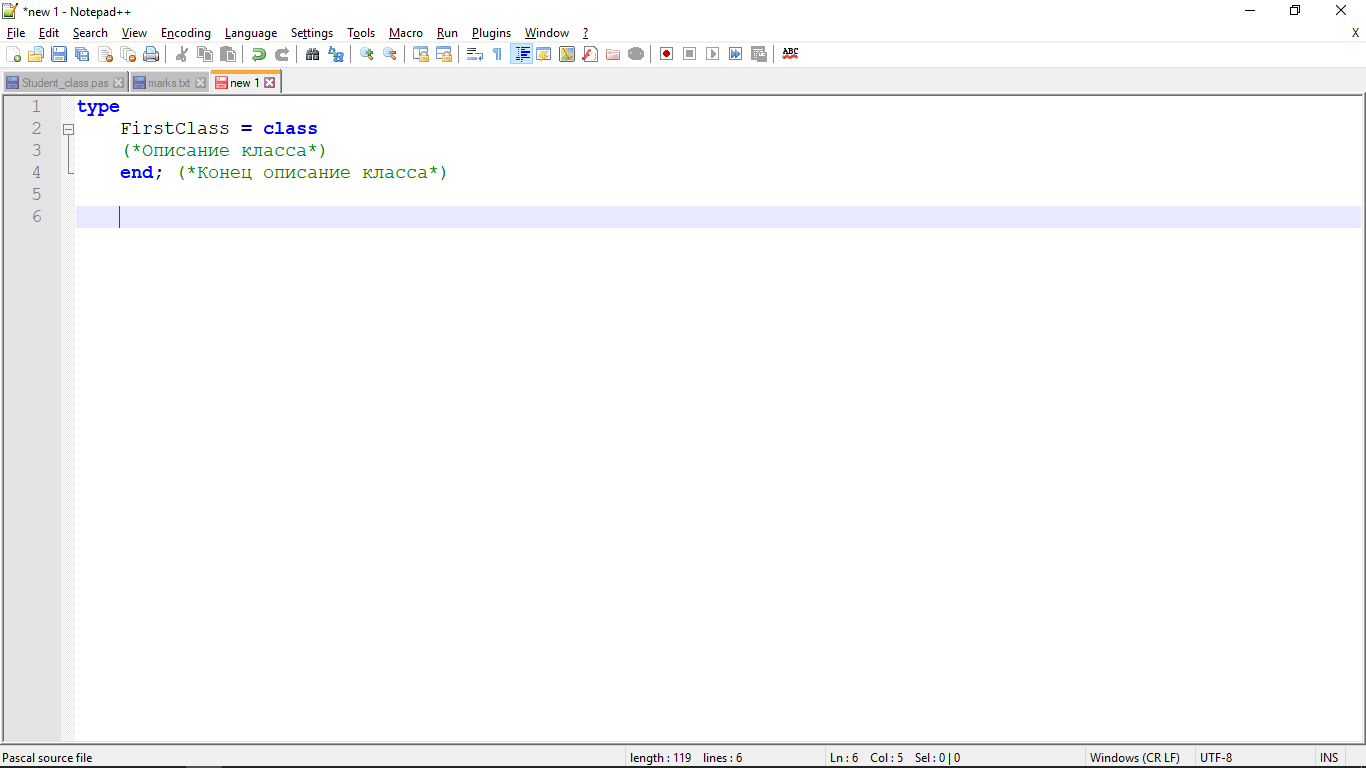


Рисунок 1

Атрибуты описываются как обычные переменные (Рисунок 2):

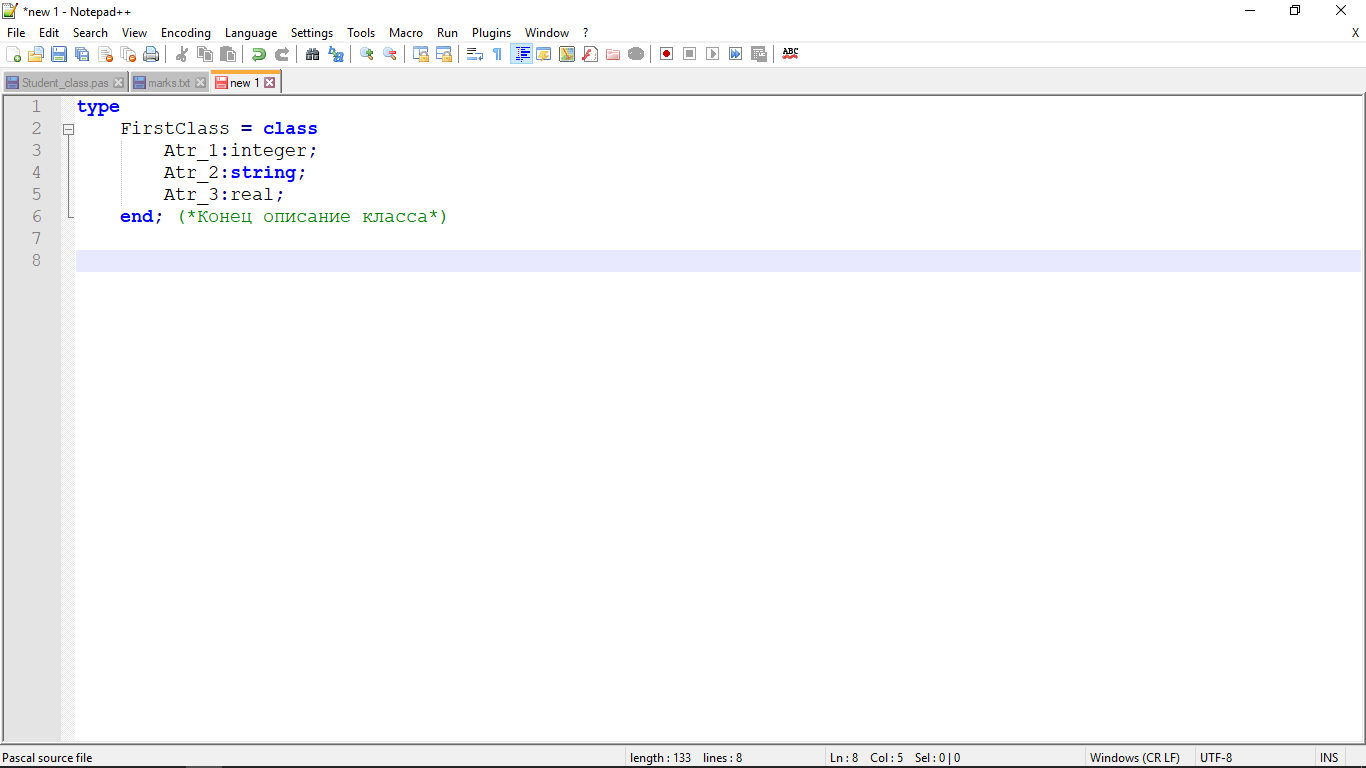


Рисунок 2

Методы описываются в секции описания класса, а его реализация задаётся после в виде “procedure\functionИмяКласса.ИмяМетода(ПередаваемыеДанные)”

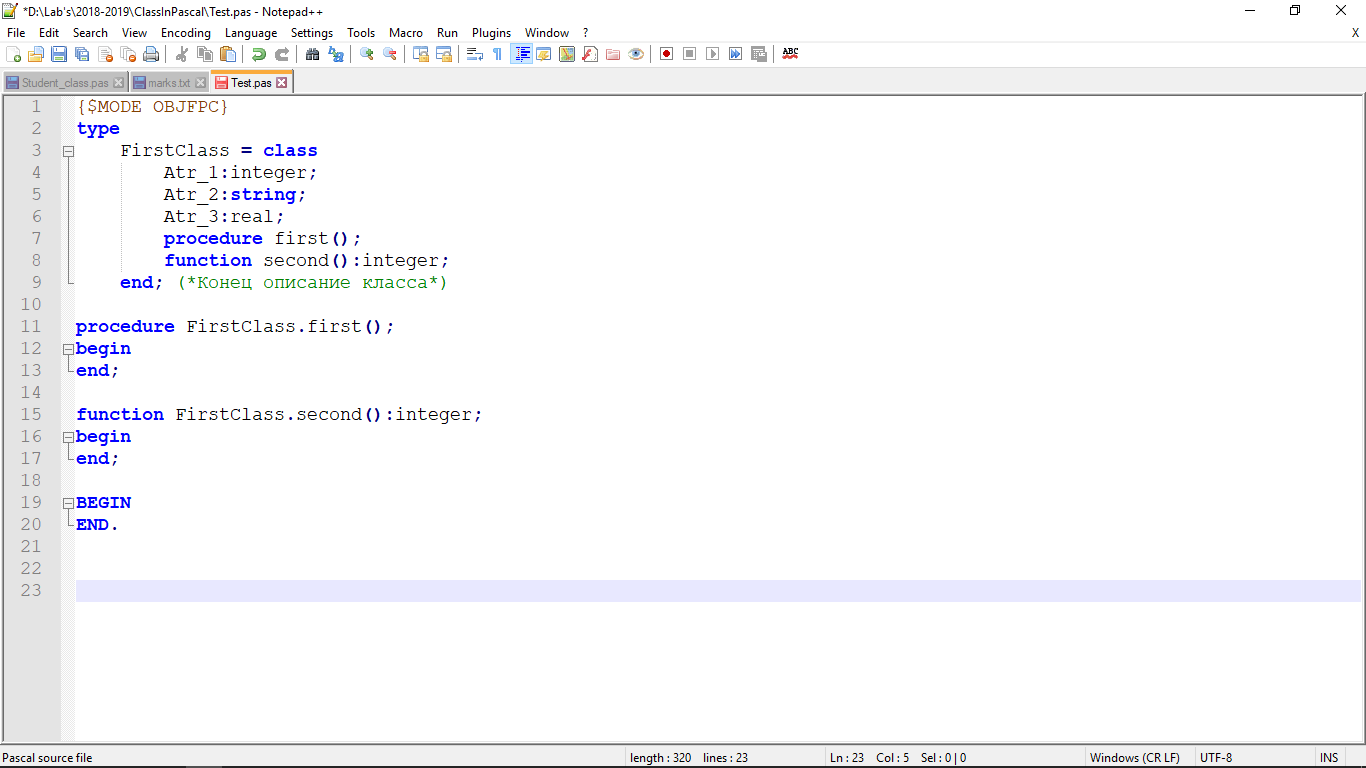


Рисунок 3

Для создания конкретного объекта в FreePascal используются специальный конструктор Create (про конструкторы и деструкторы будет сказано позже), его можно переопределить используя конструкцию constructor Create(), переопределение используется для определения атрибутов во время создания объекта.

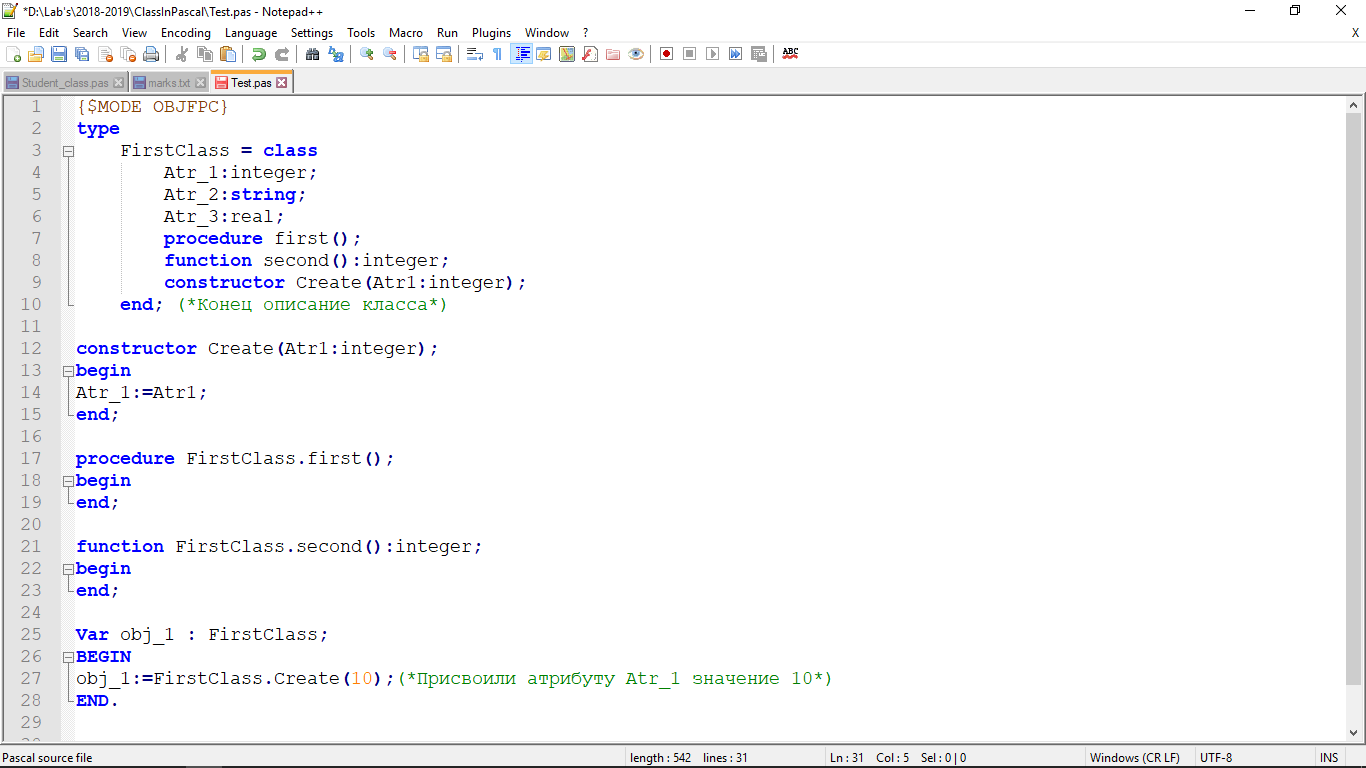


Рисунок 4

**Пример:**Допустим мы создаем программный продукт, позволяющий обрабатывать информацию о каждом студенте некоторого университета. Для этого в начале определим абстрактный класс – студент. Этот класс является общей структурой для каждого студента, в нем мы опишем: ID (уникальный номер) Студента, Имя, Фамилию, Пол, Средний балл. Каждый объект этого класса, будет являться информацией об одном студенте.  
Допустим, что информация такая как ID, Имя, Фамилию, Пол будет передаваться во время создания объекта, а информация об оценках, полученных по результатам сессии, будет храниться в отдельном текстовом файле (в файле запись вида –“ ID студента --- [оценки за сессию]”). Для того что бы вычислить средний балл для каждого студента необходимо прочитать файл, найти необходимую нам строку (содержащую ID необходимого студента), подсчитать средний балл. Именно для этой задачи будем использовать метод класса. Для класса Студент описан метода НайтиСредБалл(), который высчитывает средний балл из текстового файла. Находит оценки в файле исходя из ID студента (объекта, класса Студент). Допустим, Студент\_1 – объект класса Студент, имеющий следующие атрибуты: ID - 1, Имя – Иван, Фамилия – Иванов, Пол – М, Средний балл – “-1”(-1 – ещё не подсчитан). Для вызова метода используем конструкцию “ИмяОбъекта.ИмяМетода()” – Студент\_1.НайтиСредБалл(). Данный метод берёт информацию об ID в объекте Студент\_1, ищет в файле его оценки и находит Средний балл.

Теперь перейдём к реализации этой же задачи на FreePascal

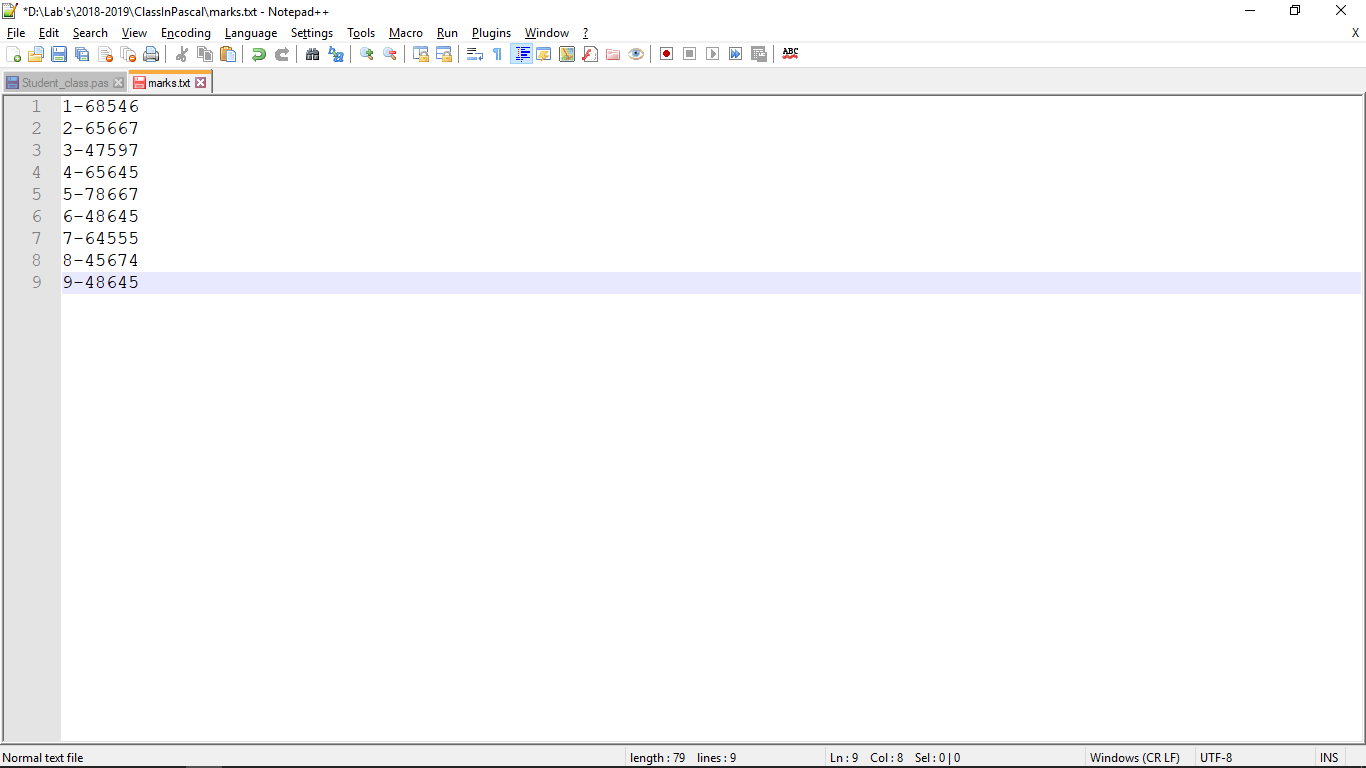


Рисунок 5

На Рис.5 представлен текстовый файл, содержащий ID студента и его оценки

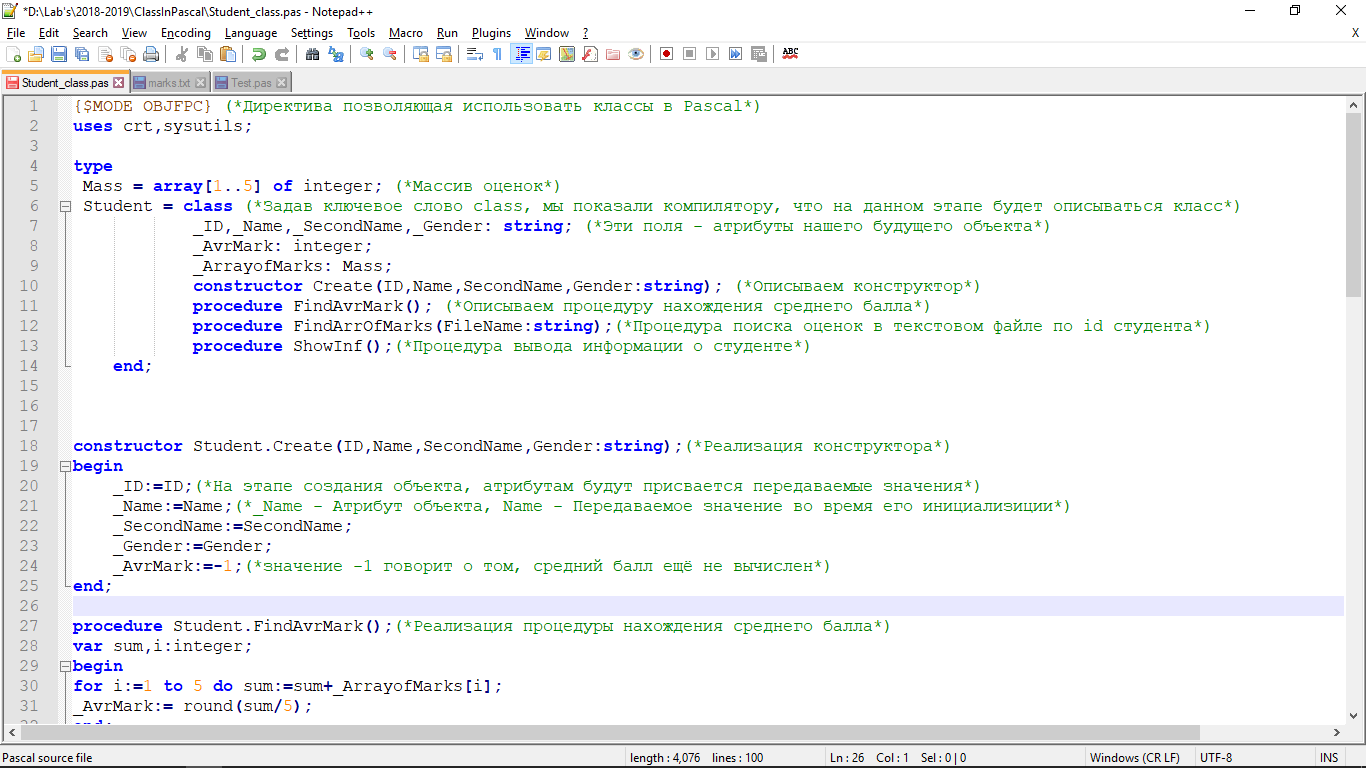


Рисунок 6

Описываем класс студент, задаем атрибуты:ID(\_ID), Имя(\_Name), Фамилия(\_SecondName), Пол(\_Gender), Средний балл(\_AvrMark) и все оценки за сессию(\_ArrayofMarks). Описываем конструктор “Create”, передавая ему: ID, Имя, Фамилию, Пол. Так же приписываем классу “Student” методы нахождения среднего балла(FindAvrMark), поиска оценок студента по ID в текстовом файле(FindArrofMarks), вывод информации о студенте(ShowInf). При реализации задачи допустим, что у каждого студента уникальный ID и у каждого студента 5 оценок за сессию. Файл с названием “marks.txt” расположен в той же директории, что и программа, его содержимое описано на Рисунке 5.

Реализация конструктора Create (Рис.7):

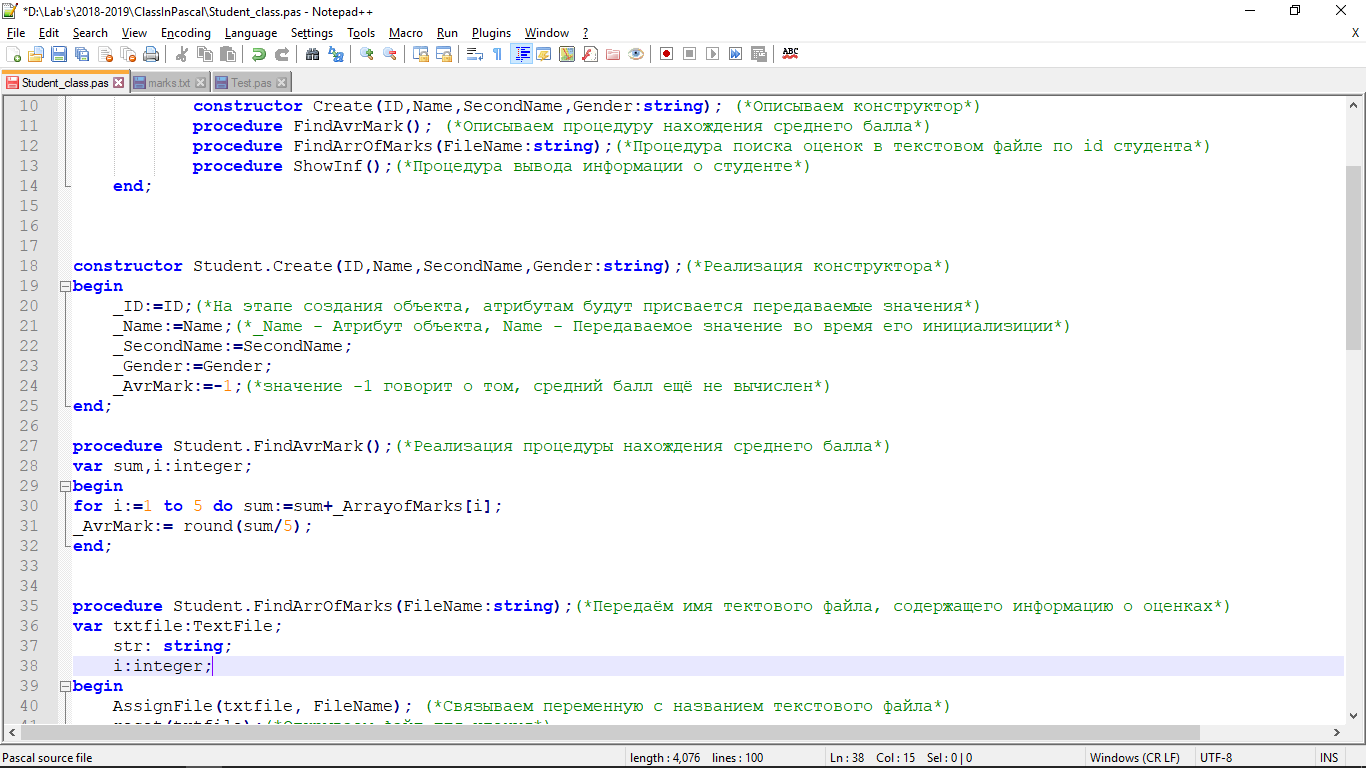


Рисунок 7

Во время описания реализация методов класса его имя задается через обращение к классу, которому метод принадлежит и имя самого метода (ИмяКласса.ИмяМетода).

Процедура поиска среднего балла (Рис.8):

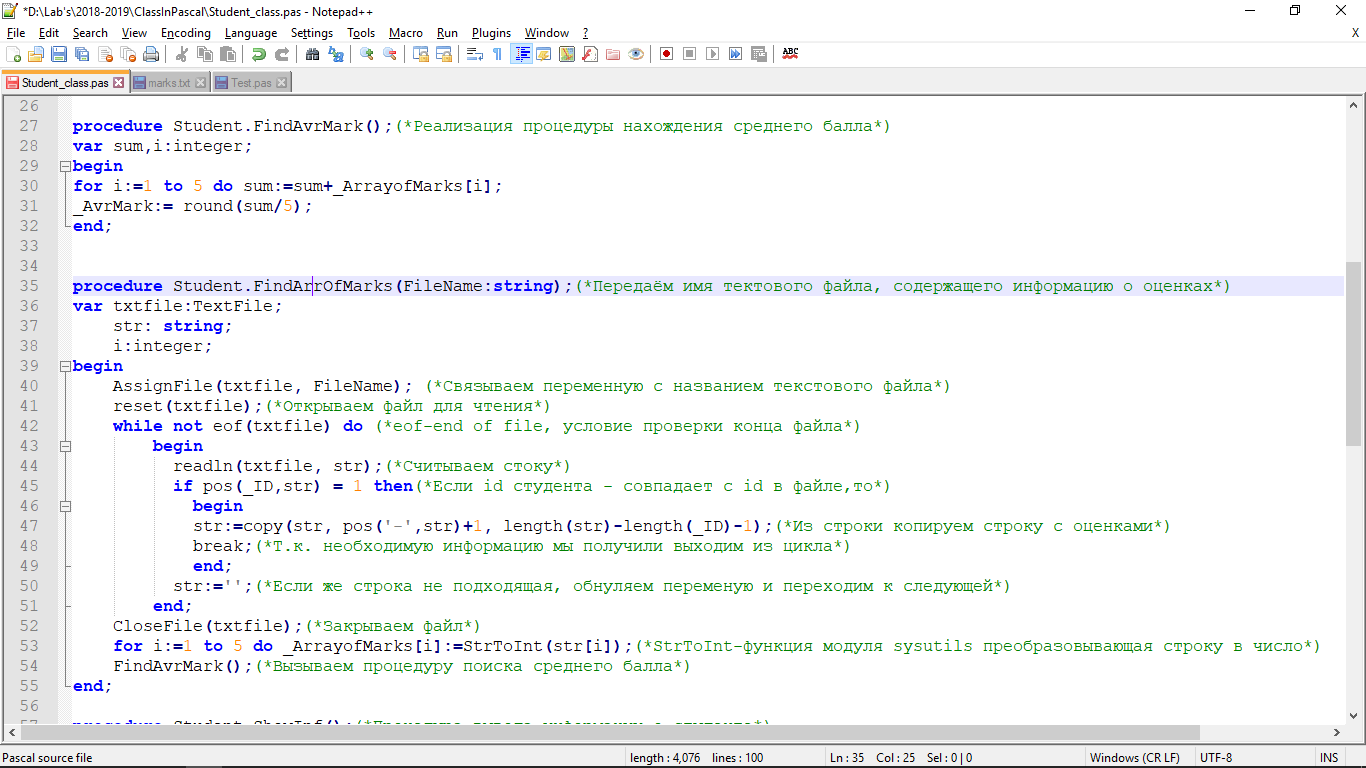


Рисунок 8

Процедура поиска оценок по ID студента в файле (Рис.9):

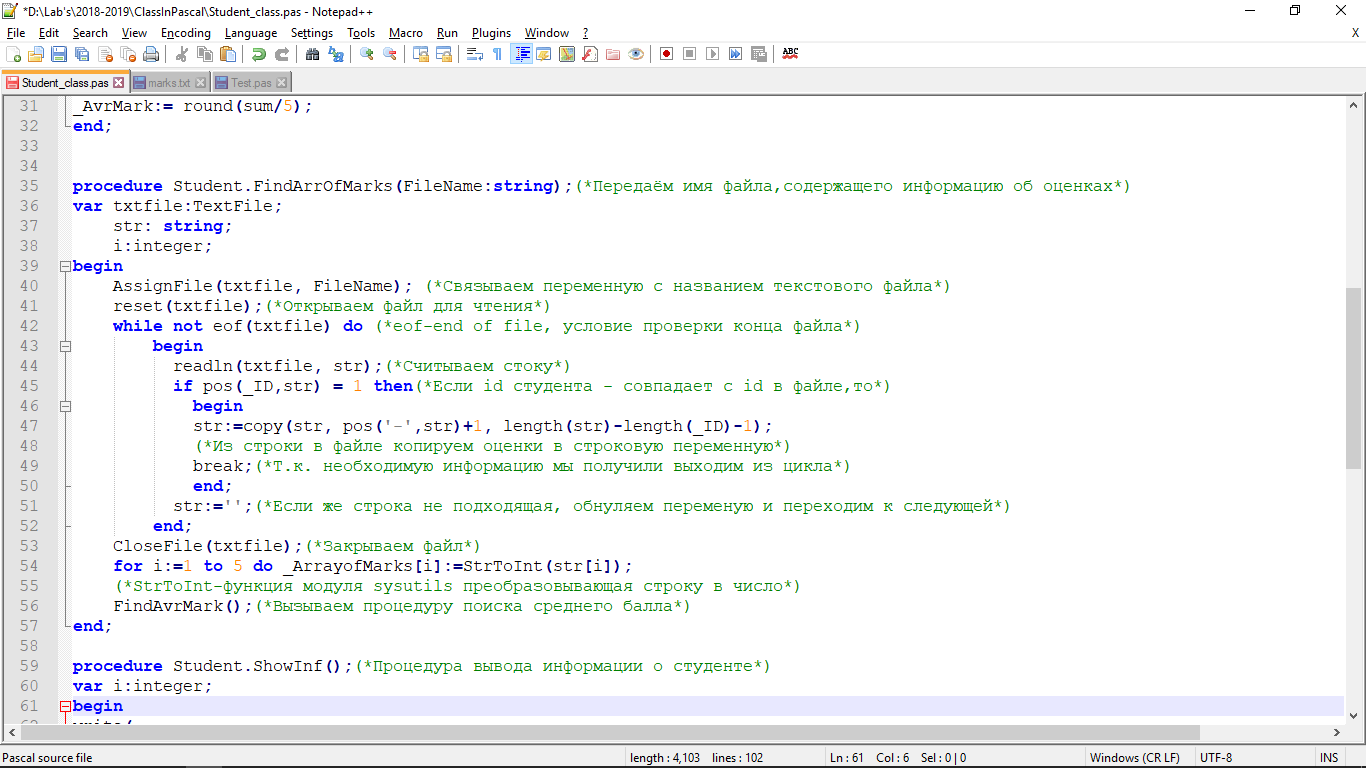


Рисунок 9

Процедура вывода информации о студенте (Рис.10):

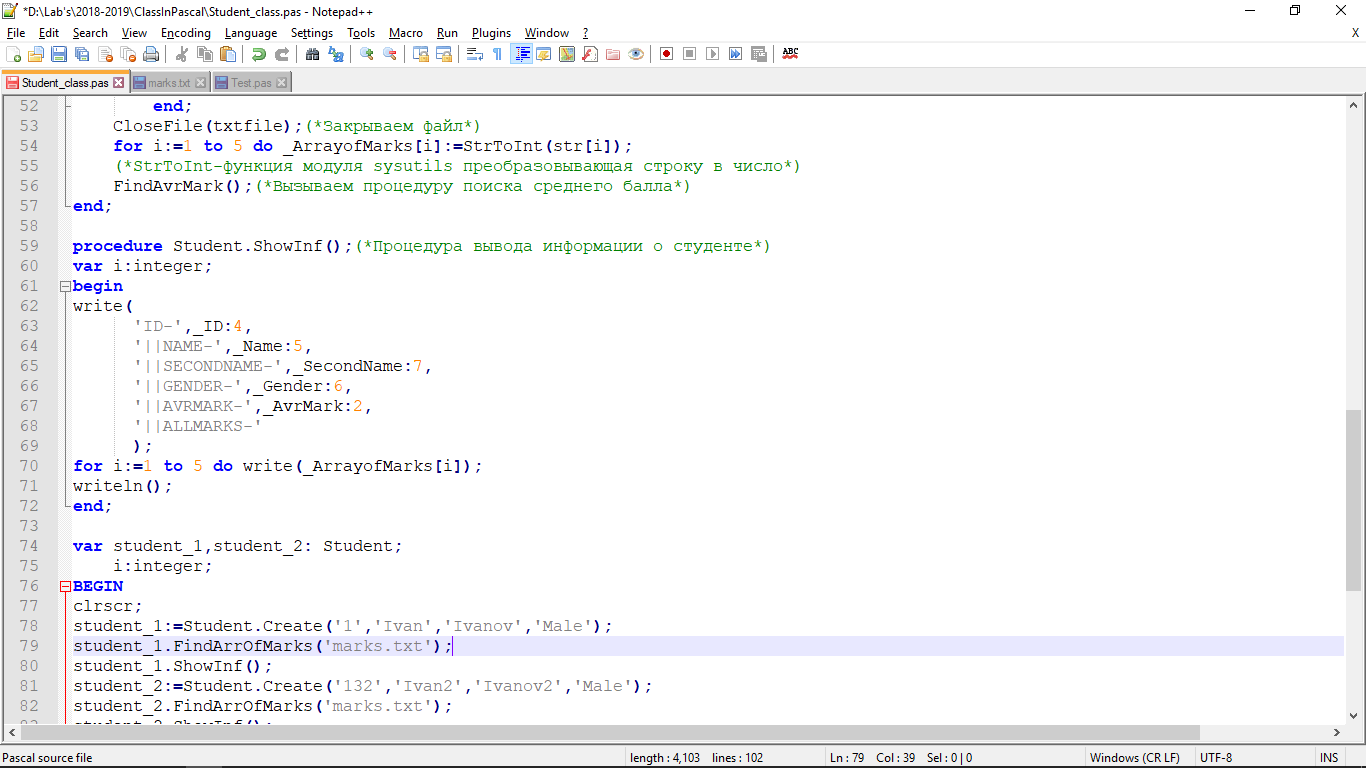


Рисунок 10

Код основной программы (Рис.11):

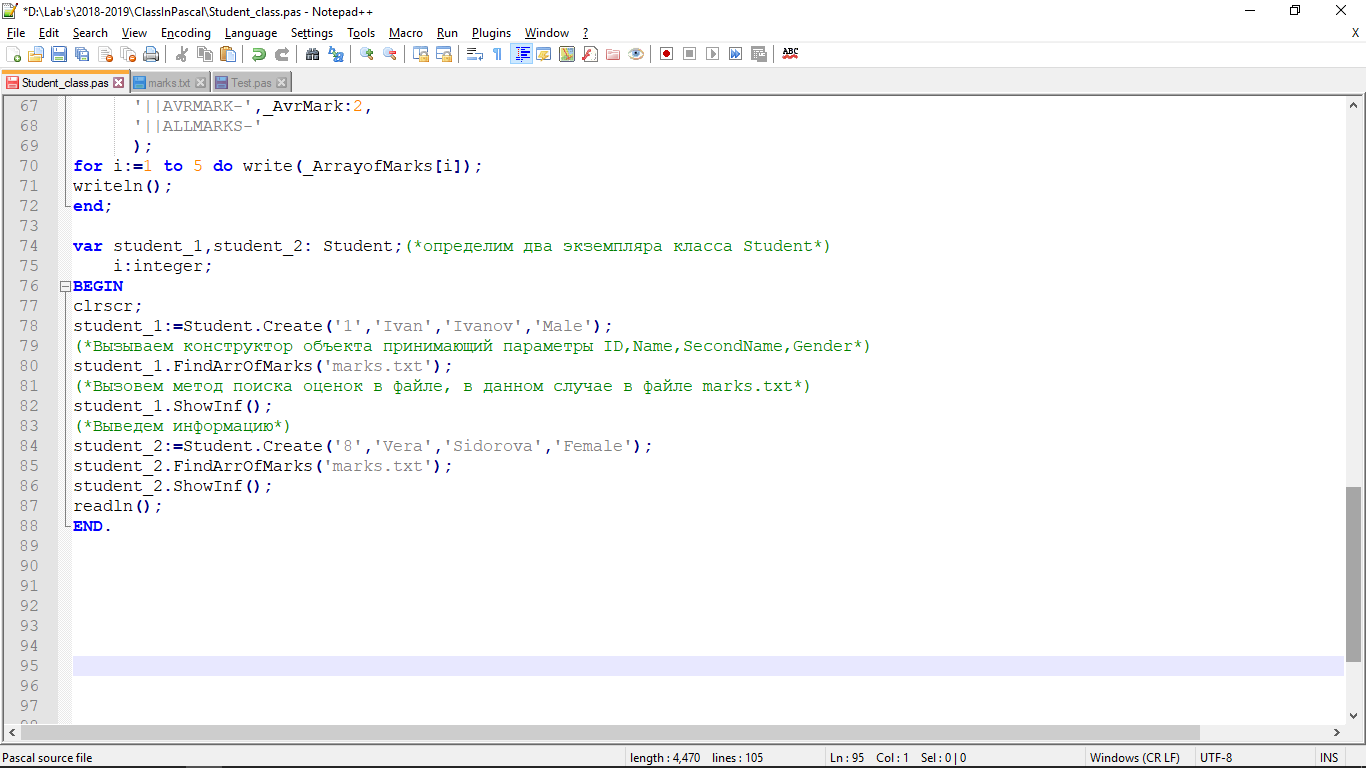
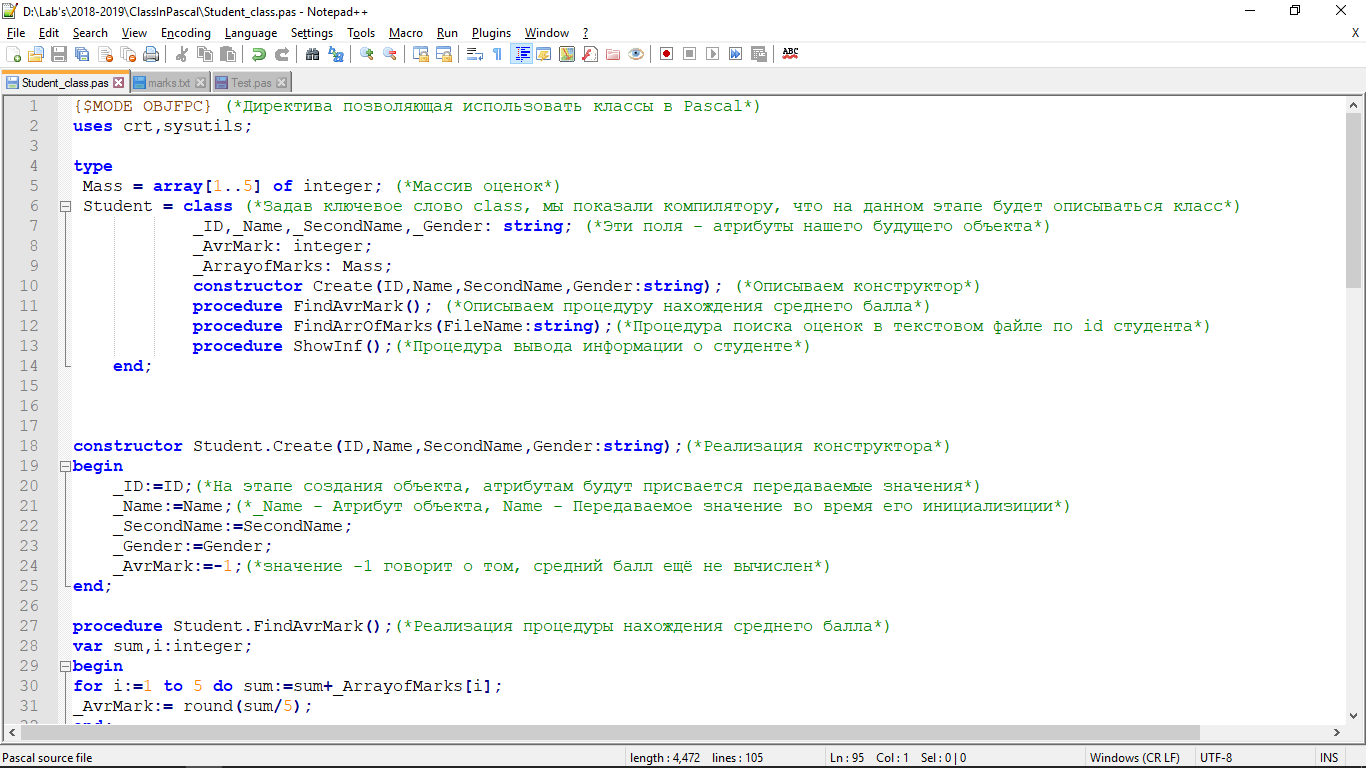
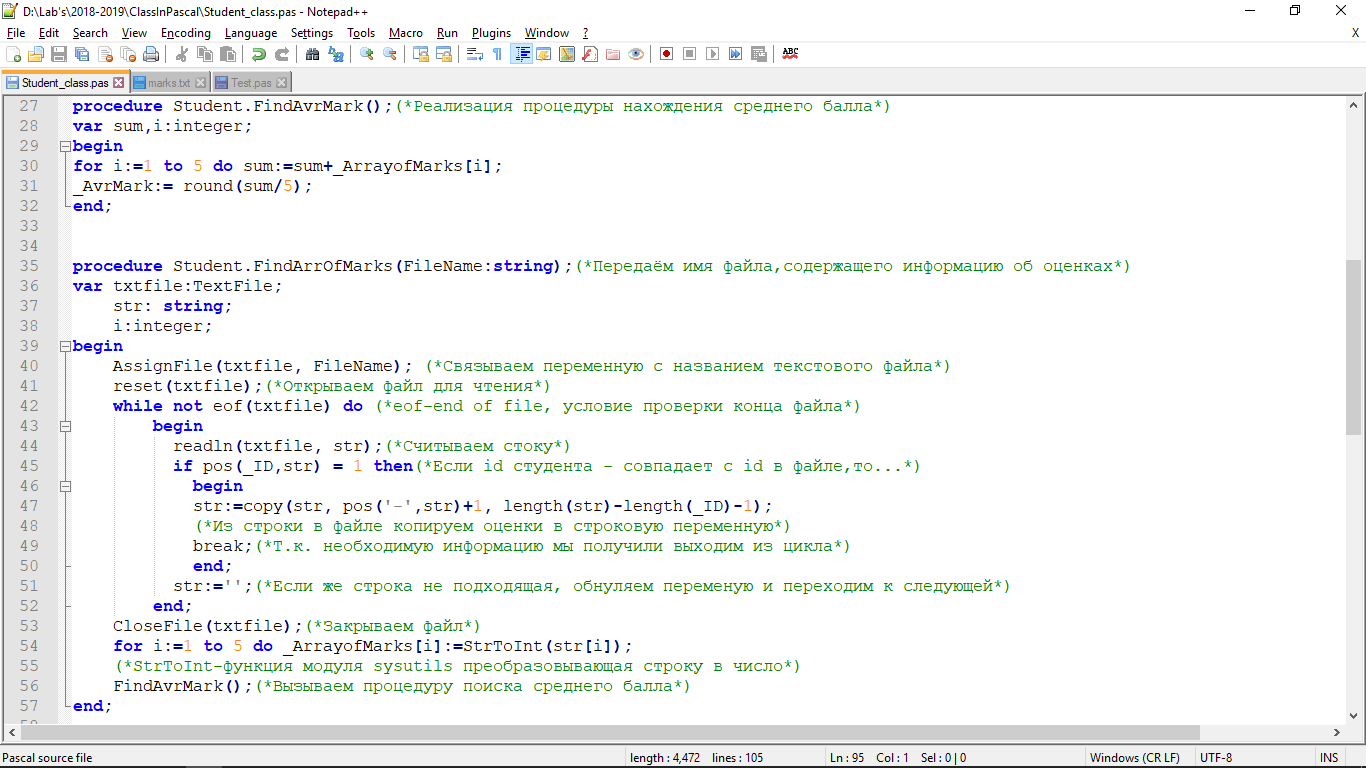


Рисунок 11

Весь код программы (Рис.12):





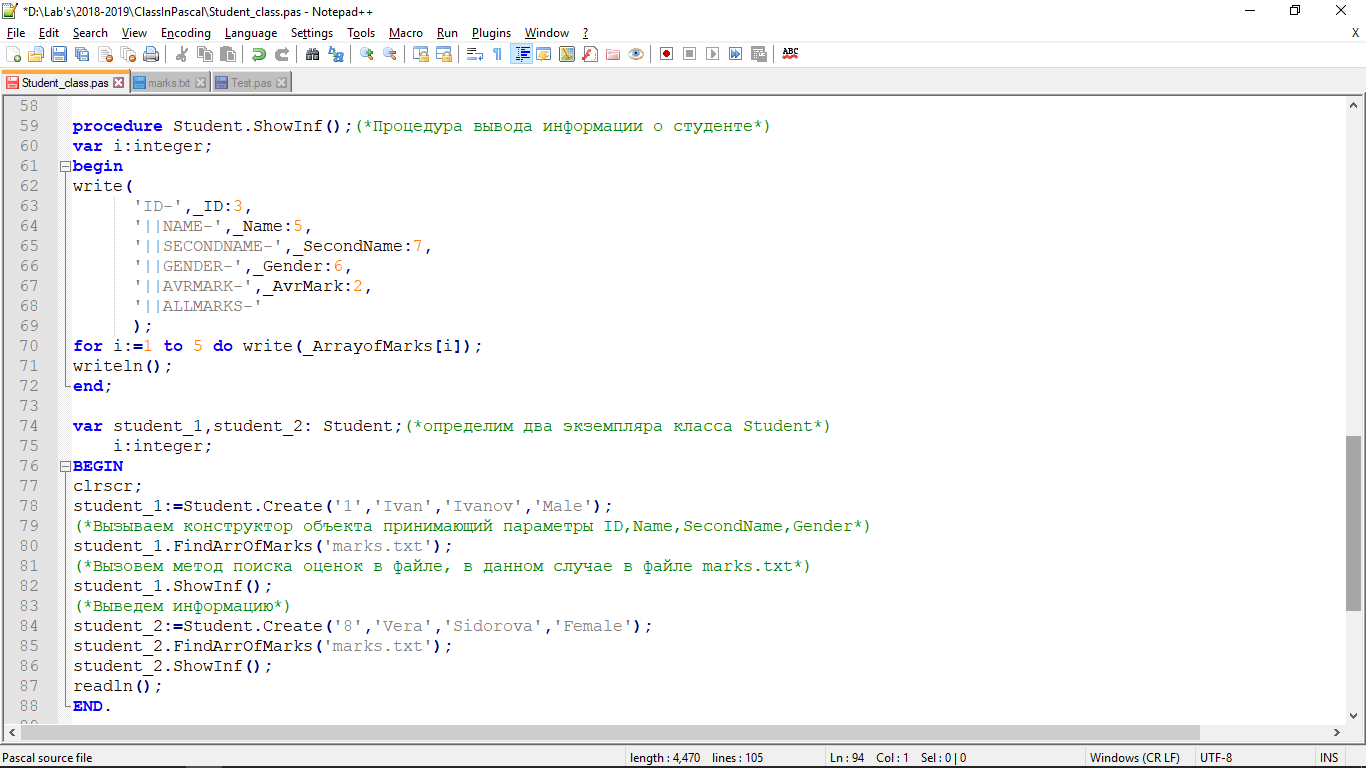


Рисунок 12

Результат выполнения программы (Рис.13):

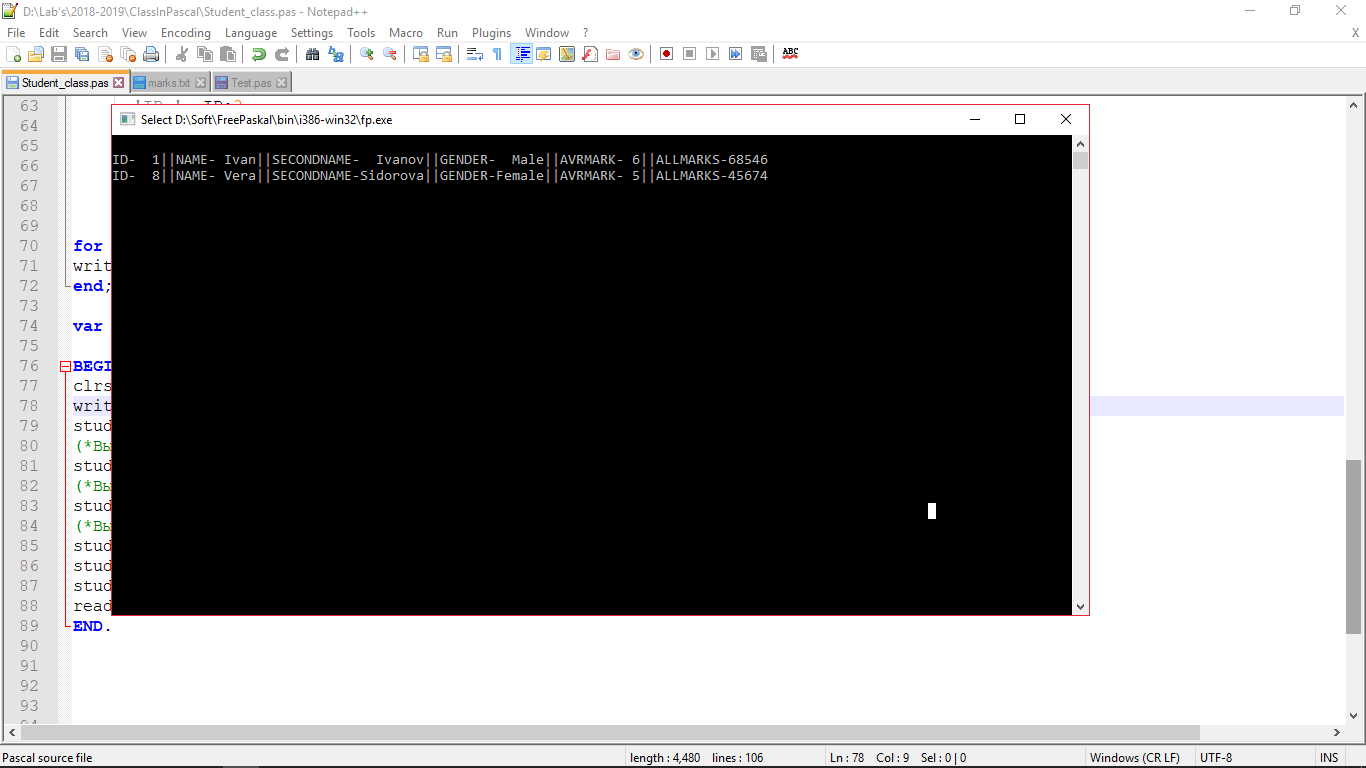


Рисунок 13

Абстрактные типы данных

Хоть термины такие как типы данных, структуры данных и абстрактные типы звучат похоже, но на деле имеют разный смысл. В языках программирования тип данных переменной обозначает множество значений, которые может принимать эта переменная. Для примера, переменная логического (булевого) типа может принимать два значения: значение истина (true) и значение ложь (false) и больше никаких других. Набор базовых типов данных отличается в различных языках программирования: в Pascal - это типы целых – integer, действительных чисел - real, булев тип - boolean и символьный тип - char. Абстрактный тип данных (АТД) – это некоторая математическая модель в составе с различными операторами, определенные в рамках этой модели. Алгоритмы могут разрабатываться в терминах АТД, но для того, чтобы реализовать алгоритм в каком-то определенном языке программирования нужно найти способ представления абстрактного типа данных в терминах типов данных и операторов, поддерживаемых данным языком программирования. Для представления абстрактных типов данных используются структуры, которые являются набором переменных, допустимо, различных типов данных, объединенных некоторым определенным образом. Базовым блоком структуры данных определяется ячейка, которая используется для хранения определенного значения составного или базового типа данных. Структуры данных описываются путем задания имен совокупностям (агрегатам) ячеек и (опционально) интерпретации значения некоторых ячеек как указателей (представителей) других ячеек. Как простейший механизм агрегирования ячеек в языке Pascal и в множестве других языков программирования возможно применение одномерного массивов, т. е. некоторой последовательности ячеек определенного типа. Массив возможно определит в качестве отображения множества индексов (например, целые числа) в множество ячеек. Обычно ссылки на ячейки состоят из имен массива и их значения из множества индексов определенного массива. В языке программирования Pascal множество индексов может являться нечисловой тип или интервальный (например 1..10). Все значения элементов массива должны иметь одинаковый тип данных. Пример объявления массива на языке Pascal: array[ТипИндекса]of ТипЯчеек; первым определяется имя для последовательности элементов, далее тип данных для индексов и, в конце, тип содержимого ячеек. Типы данных, структуры данных и абстрактные типы данных в качестве индексов массива можно было использовать буквы, надо все равно использовать целые индексы, заменяя «А» на 1, «В» на 2, и т. д. Структура записи – это ещё один механизм агрегирования ячеек в языках программирования. Запись (record) можно воспринимать как ячейку, включающую в себя нескольких других ячеек (называемых полями), их значения могут быть разных типов. Обычно записи группируются в массивы, тип данных описывается множеством типов полей записи. Третий метод агрегирования ячеек, возможный в языке Pascal и многих современных языках программирования, есть файл. Последовательность значений, некоторого определенного типа, как и одномерный массив, является файл. Но он не имеет индексов, доступ осуществляется только в том порядке, в котором они были записаны в файл. Основное различие между файлом и массивом (или записью) - это структура с произвольным доступом, (время доступа к элементам массива или записи не зависит от значения индекса массива или указателя поля записи). Плюсом использования файла заключается в том, что у файла отсутствует ограничения на количество элементов. Взглянем на роль абстрактных типов данный в процессе разработки программ, прежде чем рассмотреть абстрактный тип данных. Для начала, сравниваем АТД с процедурой. Процедура – некоторый интегральный инструмент программирования, которое возможно рассмотреть, как более общее понятие оператора. В сравнении с встроенными операторами языков, ограниченного возможностями, посредством процедур, возможно описать собственные операторы и применить их к операндам различных типов, не только основных. Подпрограмма перемножения матриц - пример такого оператора процедуры. Результат действия оператора может также иметь тип кроме определенного в АТД эта модель. Две характеристики процедур – обобщение и инкапсуляция, – о котором будет сказано позже, отлично характеризуют абстрактный тип данных. АТД можно рассмотреть, как обобщение простых типов данных, таким же образом поскольку процедура является обобщением простых операторов (\*, /, …). Если необходимо изменить реализацию АТД, мы знаем, где найти и что измениться в одном маленьком разделе программы, и мы можем быть уверены, что это не приведет к ошибкам куда-нибудь в программе во время работы с этим типом данных. Кроме того, из раздела с определением операторов АТД возможно рассмотреть типы АТД как основные типы, поскольку объявление типов официально не соединено с их реализацией. Термин реализация АТД означает следующее: перевод в операторах языка программирования объявлений, определяющих переменные этого абстрактного типа данных, плюс процедура каждого оператора, кто выполняется по объектам АТД. Реализация зависит от структуры данных, АТД представления. Каждая структура данных строится на основе базовых типов данных применяемого языка программирования, используя доступные в этом языке средства структурирования данных. Структуры массивов и записей – два важных средства структурирования данных, возможных на языке Паскаль. Одна из основных причин определения двух различных АТД в рамках одной модели - то, что по объектам их АТД необходимо выполнить различные операции, т.е. определить операторы различных типов. Желательно записать программы в языке, базовых типах данных и какие операторы достаточно для реализации АТД.

**Абстрактные классы**

*Абстрактный класс* — это класс, содержащий один или несколько методов, которые не имеют какой-либо обеспеченной реализации. Представим, вы описали класс Shapе(Форма). Этот класс – абстрактный, т.к. нельзя создать его экземпляр. Если вас попросят нарисовать фигуру, то вы спросите: “Нарисовать фигуру какой формы?”. В этом случае, концепция фигуры будет абстрактной. Квадрат, допустим, - это уже конкретная концепция, вы знаете как её нарисовать. При использовании абстракции в нашей программе, синтаксис для рисования был один и тот же. Любой класс, описывающий фигуры и описанный в нашей программе, должен иметь метод с именем draw(), выводящий на экран фигуру, вне зависимости от его вида. Это позволяет сократить время на описание этого метода для каждого класса, а также снижает количество синтаксических ошибок.

Инкапсуляция

Инкапсуляция (encapsulation) - механизм, объединяющий данные и код, защищающий от внешнего вмешательства или неправильного использования и манипулирующий этими данными. В объектно-ориентированном программировании код и данные могут быть объединены вместе; Тогда говорят, что создаётся чёрный ящик. В случае, если коды и данные объединяются таким способом, создаётся объект (object). Другими словами, объект - это то, что поддерживает инкапсуляцию.  
Коды и данные могут быть закрытыми (private) в объекте. Так называемый Privet блок доступен только для частей этого определенного объекта. Так получаем, что данные объекта недоступны для других частей программы, которые описаны вне объекта. Если описание объекта являются открытым, вне зависимости от того, что они указаны(описаны) внутри объекта, они будут доступны для других областей программы. Обычно открытая область объекта используется в качестве взаимодействия с закрытыми элементами объекта.  
На самом деле объект является переменной определённого пользователем типа. Может показаться странным, что объект, который объединяет коды и данные, можно рассматривать как переменную. Однако применительно к объектно-ориентированному программированию это именно так. Каждый элемент данных такого типа является составной переменной.

Основной плюс использования объектов заключается в том, что ему нет необходимости показывать все свои поля и методы. При хорошем проектировании программы с использованием ООП объект должен показывать только те интерфейсы, которые нужны иным объектам для взаимодействия с ним. Остальные данные, не относящиеся к взаимодействию с экземпляром класса, должны быть скрыты от всех других объектов.   
Инкапсуляция определяется тем, что объекты содержат как атрибуты, так и по­ведения. Скрытие данных является основной частью инкапсуляции. Инкапсуляция и скрытие данных

**Методы доступа**

В большинстве, если не во всех примерах, приведенных в этой книге, атрибуты определяются как private, из-за чего у любых других объектов нет прямого досту­па к этим атрибутам. Было бы глупо создавать объект в изоляции, которая не по­зволит ему взаимодействовать с другими объектами, — мы ведь хотим, чтобы он мог делиться с ними соответствующей информацией. Есть ли необходимость ин­спектировать и иногда изменять значения атрибутов других классов? Ответом на этот вопрос будет, конечно же, «да». Бывает много ситуаций, когда тому или иному объекту требуется доступ к атрибутам другого объекта; однако это не обязательно должен быть прямой доступ.

Класс должен очень хорошо защищать свои атрибуты. Например, вы не захо­тите, чтобы у объекта А была возможность инспектировать или изменять значения атрибутов объекта В, если объект В не сможет при этом все контролировать. На это есть несколько причин, и большинство из них сводится к целостности данных и эффективной отладке.

Предположим, что в классе Cab есть дефект. Вы отследили проблему, что при­вело вас к атрибуту name. Каким-то образом он перезаписывается, а при выполнении некоторых запросов имен появляется мусор. Если бы name был public и любой класс мог изменять его значение, то вам пришлось бы просмотреть весь возможный код в попытке найти фрагменты, которые ссылаются на name и изменяют его значение. Однако если бы вы разрешили только объекту Cabbie изменять значение name, то вам пришлось бы выполнять поиск только в классе Cabbie. Такой доступ обеспечивает­ся методом особого типа, называемым *методом доступа*. Иногда методы доступа называются геттерами и сеттерами, а порой — просто get() и set(). По соглашению в этой книге мы указываем методы с префиксами set и get, как показано далее:

В этом фрагменте кода объект Supervisor должен отправить запрос объекту Cabbie на возврат значения его атрибута name (рис. 4.4). Важно здесь то, что объект Supervisor сам по себе не сможет извлечь информацию; ему придется запросить сведения у объекта Cabbie. Эта концепция важна на многих уровнях. Например, у вас мог бы иметься метод setAge(), который проверяет, является ли введенное значение возраста 0 или меньшей величиной. Если значение возраста окажется меньше 0, то метод setAge() может отказаться задавать это некорректное значение. Обычно сеттеры применяются для обеспечения того или иного уровня целост­ности данных.

Это также проблема безопасности. Например, у вас имеется требующая защиты информация: пароли или данные расчета заработной платы, доступ к которым вы хотите контролировать. Таким образом, доступ к данным с помощью геттеров и сет­теров обеспечивает возможность использования механизмов вроде проверки паролей и прочих методик валидации. Это значительно укрепляет целостность данных.

**Реализация на Pascal:**

Name\_of\_class= class

Private  
 <поля и методы, доступные только в пределах модуля>  
protected  
 <поля и методы, доступные только в классах потомках>  
public  
 <поля и методы, доступные из других модулей>  
published  
 <поля и методы, видимые в инспекторе объектов>

End;

Опишем некоторый класс TPerson (рис.14), имеющий следующие закрытые поля: Имя – Fname, Адрес – Faddress, и следующие процедуры: конструктор –Create (здесь будет так же задаваться имя), Вывод информации – show, Функции получения имени и адреса – GetName, GetAddress соответственно, процедура задания адреса – SetAddress.

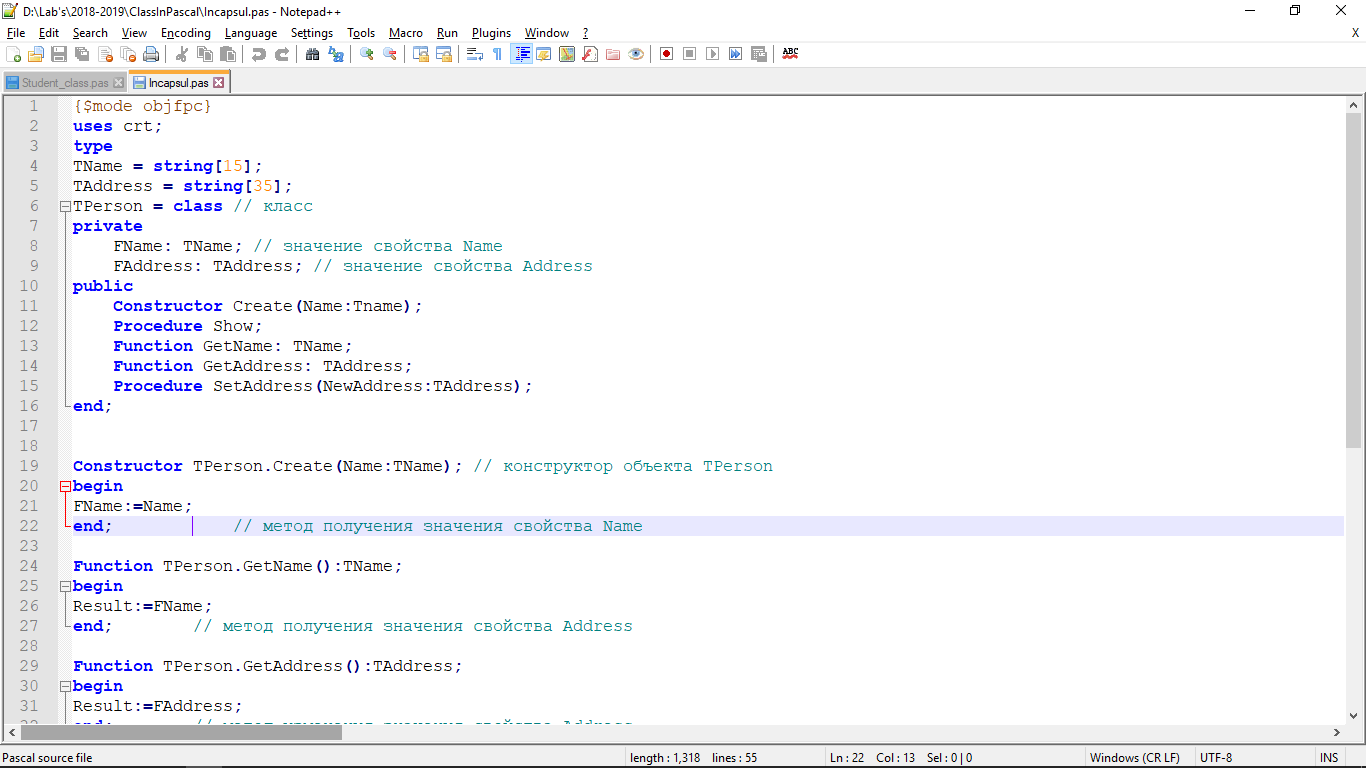


Рисунок 14

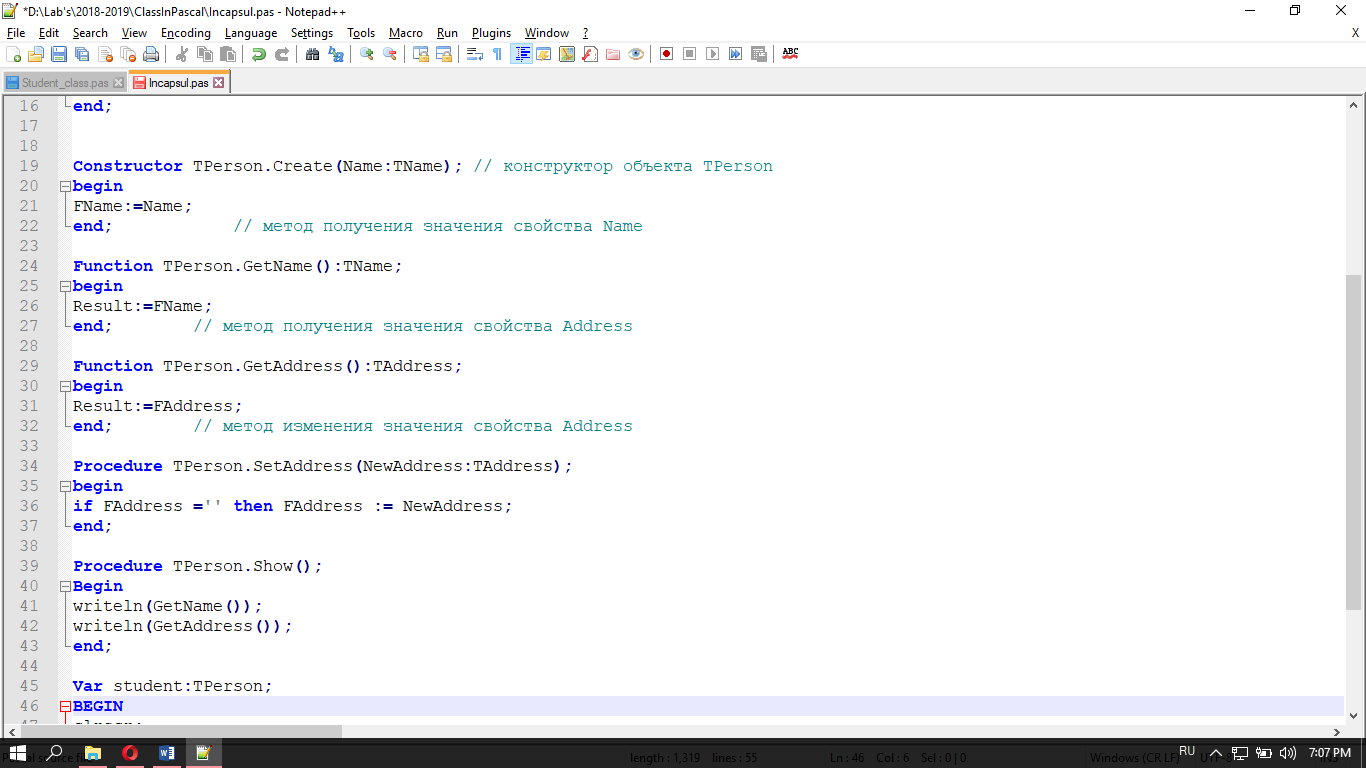
Реализация функций (рис.15):

Рисунок 15

Код основной программы (рис.16):

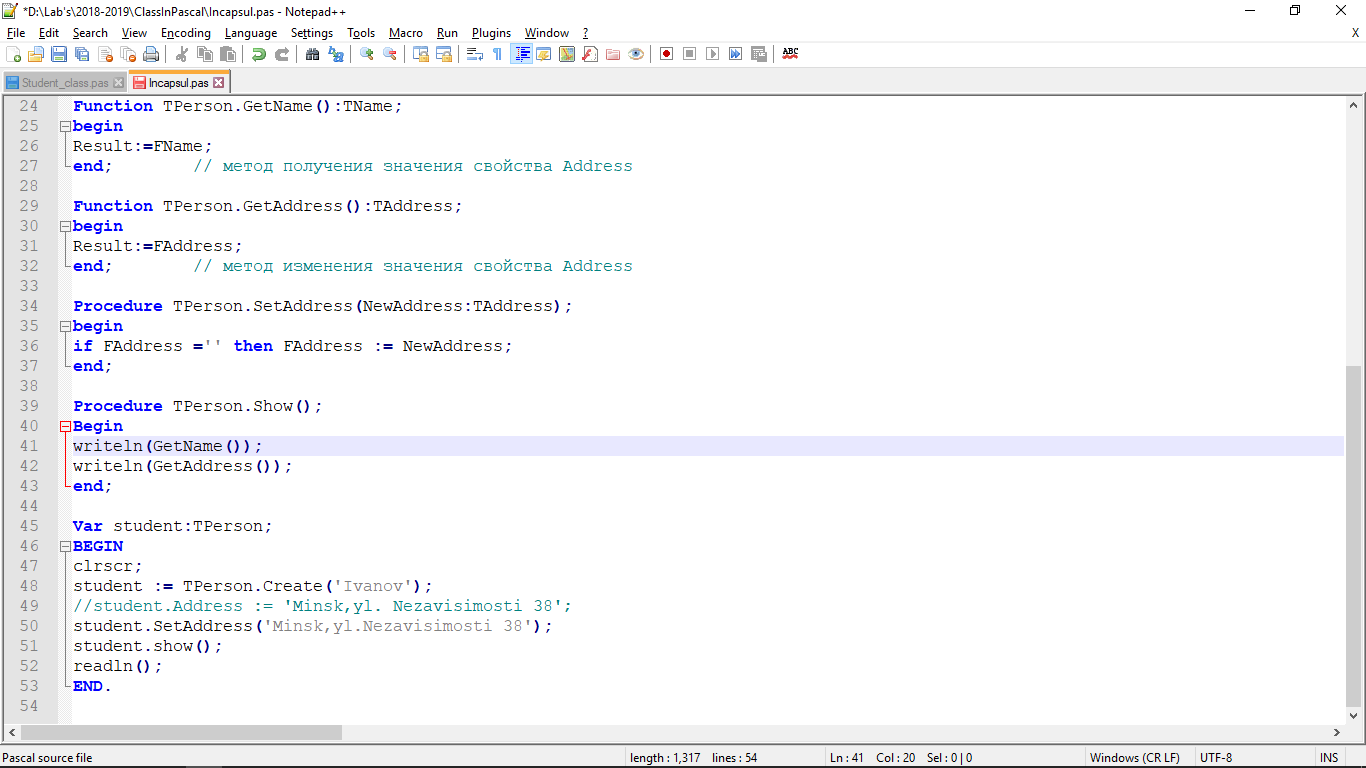
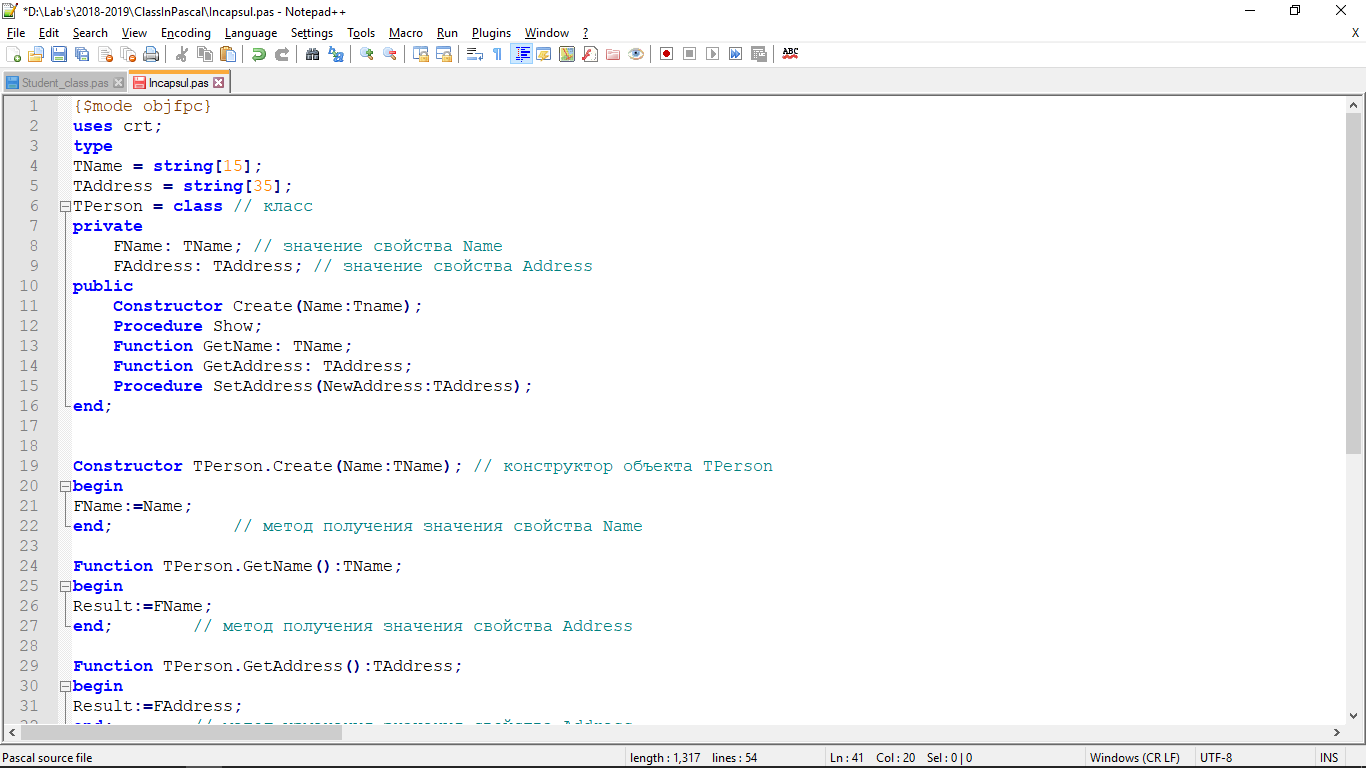


Рисунок 16

Код всей программы (рис.17):



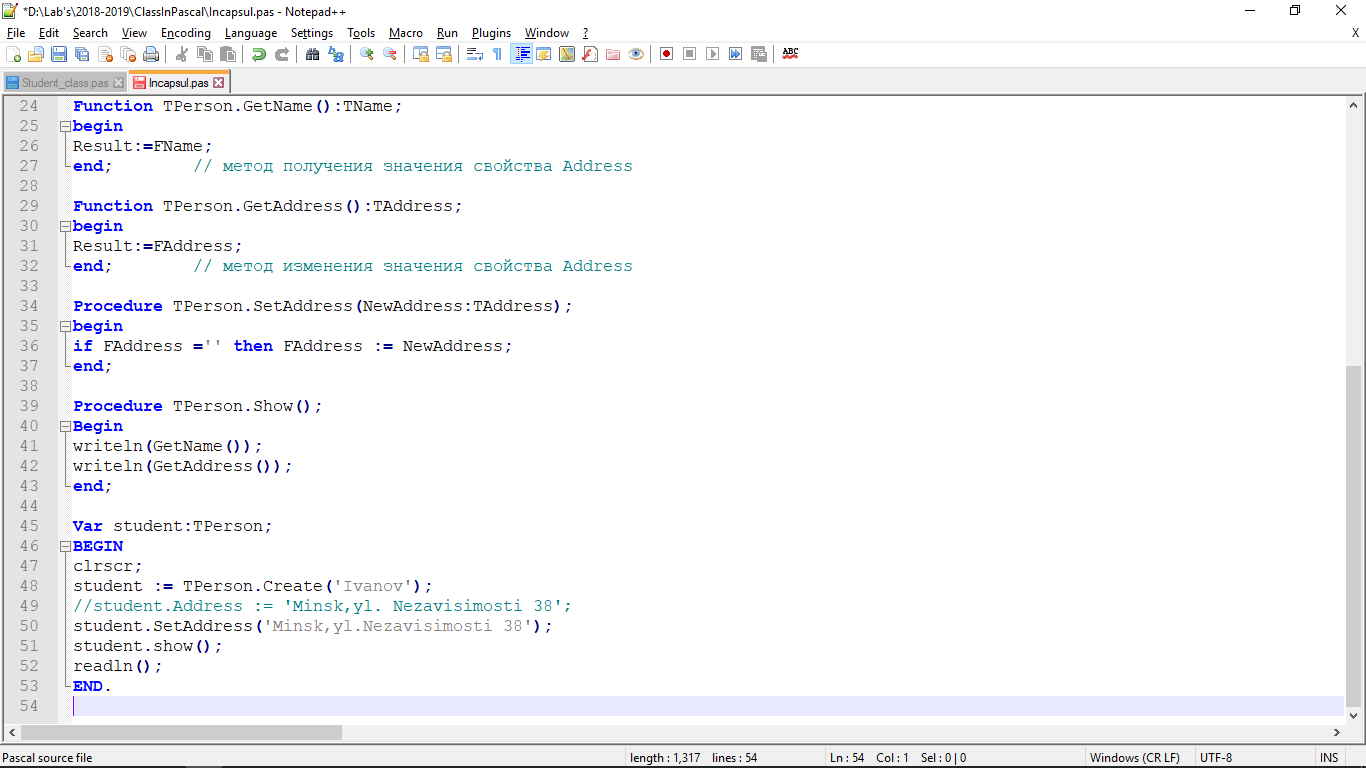


Рисунок 17

Обращение к переменной Private части (address) напрямую даст ошибку.

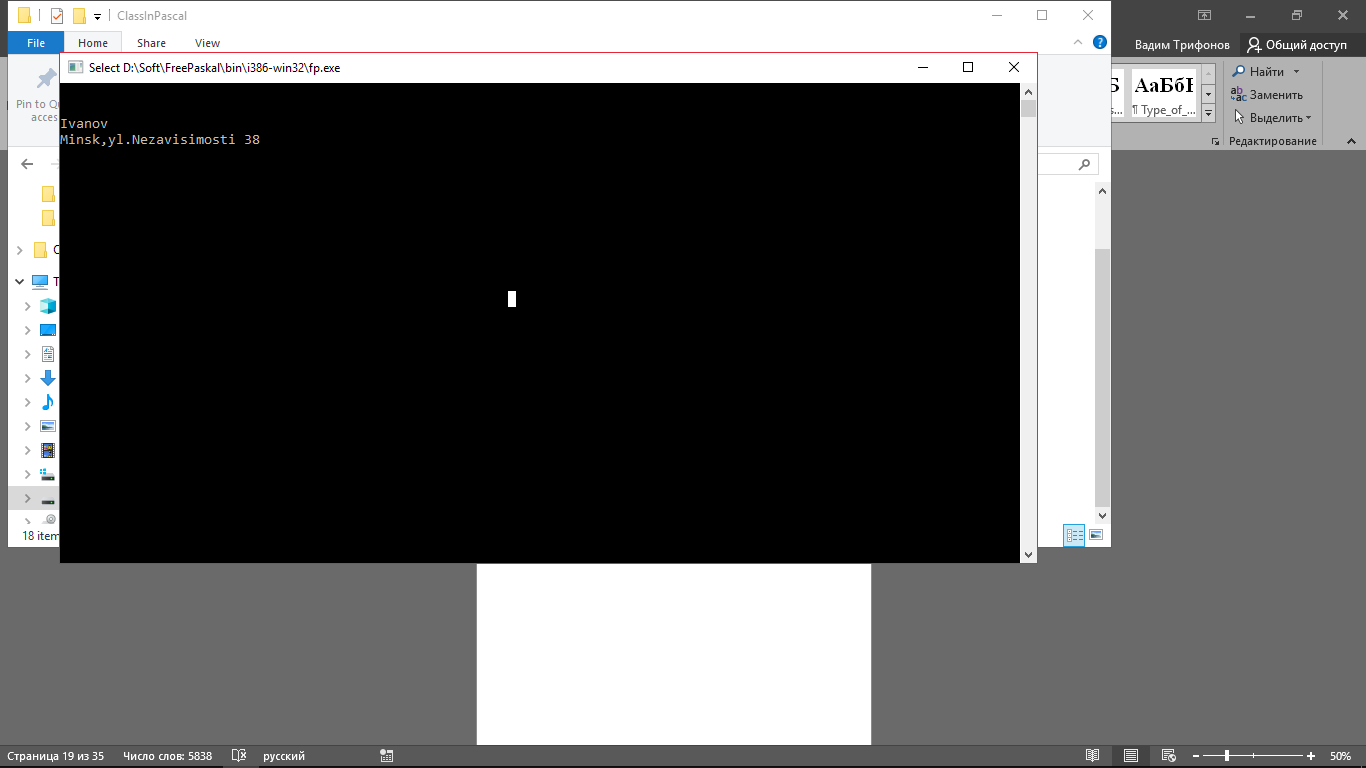
Результат выполнения программы (рис.18):  


Рисунок 18

Наследование и Полиморфизм

Сильная сторона объектно-ориентированного программирова­ния - это повторное использование кода. При структурном проек­тировании повторное использование кода допускается в известной мере: вы може­те написать процедуру, а затем применять ее столько раз, сколько пожелаете. Так же наследование применяется как одна из частей реализации абстрактных классов. Важно, что ООП делает шаг вперед, позволяя вам описывать отношения между классами, которые облегча­ют повторное использование кода и создаёт общую связанную структуру, путем упорядочения и учета базовых общностей разных классов. *Наследование* является главным средством обеспечения этой функциональности.

Возможность класса использовать атрибуты и методы другого класса и есть наследование. Такая функциональность позволяет реализовывать абсолютно новые классы путем абстрагирования некоторых общих атрибутов и методов.

Основная задача при проектировании задачи с использованием наследования является выделением общности разнообразных классов. Например, вы сконструировали класс Dоg (с англ. Собака) и класс Cаt (с англ. Кот), каждый из них будет содержать атри­бут еyеCоlоr (с англ. Цвет глаз). Процедурная модель вынуждала бы описывать атрибут eyeColor как для Dog, так и для Cat, в это же время, объектно-ориентированное проектировании, позволяет атрибут, который связан с цветом глаз, перенести в отдельный класс с именем Animal (с англ. Животное) вместе с всеми другими об­щими атрибутами и методами. При проектировании будет удобно оба класса (Dog и Cat) наследовать от класса Animal.

**Суперклассы и подклассы**

Суперкласс, родительский класс или же базовый, включает в себя все те атрибуты и методы, которые являются общими для классов, которые наследуют от него. Для примера возьмём тот же класс Animal, он будет базовым для классов Dog и Cat, они оба будут иметь атрибут eyeColor. Нет необходимости в каждом их этих классов описывать этот атрибут, ведь он является для них общим. Введем так же дополнительный метод для класса Animal – Grow (от англ. расти) Все классы, наследуемые от класса animal, будут включать в себя метод Grow и так же хорошо работать с другими экземплярами этого класса. Если же необходимо изменить поведение некоторого метода для определенного наследуемого класса есть возможность использовать Полиморфизм, который будет рассмотрен позже.

Подкласс, дочерний класс или же производный класс, являются расширением суперклассов. Вернемся к предыдущему примеру: классы Dog и Cat наследуют все общие атрибуты и методы от класса Animal. Класс Animal будет супер­классом дочерних классов Dog и Cat.

Наследование предоставляет много преимуществ в виде функциональности при проектировании программ. Возьмем класс Cat, суперкласс Animal включает в себя большую часть требуемой функциональности. Наследуя от класса Animal, Cat уже содержит все атри­буты и методы, которые делают его отдельным и самостоятельным классом, содержащего весь необходимый функционал.   
Дерево наследования может разрастись довольно сильно. Каждый дочерний класс может стать в свою очередь суперклассом для другого, например, класс Cat также может выступать в роли суперкласса, если понадобится дополнительно абстрагироваться, чтобы обеспечить классы для различных пород кошек. Одной из сильных сторон наследования являются методики абстрагирования и организации.

В множестве объектно-ориентированных языков программирования (сюда входит и Free Pascal) класс может иметь только один родительский, но много дочерних классов. А в некоторых языках программирования, например C++, у одного класса может быть несколько родительских классов. В первом случае наследование называется *простым*, а во втором — *множественным*.  
Допустим есть задача, в которой Square (от англ. квадрат), Circle (от англ. круг) и Star (от англ. звезда) наследуют от Shape. Это от­ношение называется *«является экземпляром»*, т.к. круг есть форма, как и квадрат. Когда подкласс наследует от суперкласса, он получает все возможности, которыми обладает этот суперкласс. Таким образом, Circle, Square и Star являются дочерними классами или расширениями класса Shape.

**Как сконструирован суперкласс**

При реализации задачи при помощи наследования вы должны точно знать, каким образом описан определенный суперкласс. Важно помнить, что, когда задействуется наследование, от роди­тельского класса наследуется все. Следовательно, необходимо хорошо знать все данные и методы базового класса. То как наследуются атрибуты довольно понятно, но то, как наследуются конструкторы, не так очевидно.

* Внутри конструктора происходит вызов конструктора суперкласса соответ­ствующего класса. Если явного вызова конструктора суперкласса нет, то авто­матически вызывается конструктор по умолчанию.
* Инициализируется каждый атрибут класса объекта. Эти атрибуты являются частью определения класса (переменные экземпляра).
* Выполняется остальная часть кода внутри конструктора.

*Полиморфизм* — это греческое слово, буквально означающее множественность форм. Несмотря на то что полиморфизм тесно связан с наследованием, он часто упоминается отдельно от него как одно из наиболее весомых преимуществ объектно-ориентированных технологий. Если потребуется отправить сообщение объекту, он должен располагать методом, определенным для ответа на это со­общение. В иерархии наследования все подклассы наследуют от своих супер­классов. Однако, поскольку каждый подкласс представляет собой отдельную сущность, каждому из них может потребоваться дать отдельный ответ на одно и то же сообщение.

Производный класс наследует от класса-предка поля и методы; если имеет место совпадение имен методов, то говорят, что они перегружаются. В зависимости от того, какие действия происходят при вызове, методы делятся на следующие группы: • статические методы; • виртуальные методы; • динамические методы. По умолчанию все методы статические. Эти методы полностью перегружаются в классах-потомках при их переопределении. При этом можно полностью изменить объявление метода. Если обращаться к такому методу у объекта базового класса, то будет работать метод класса-родителя. Если обращаться к методу у производного класса, то будет работать новый метод. Виртуальные и динамические методы имеют в базовом и производном классах те же имена и типы. В классах-наследниках эти методы перегружены. В зависимости от того, с каким классом работают, соответственно и вызывается метод этого класса. Основная разница между виртуальными и динамическими методами — в способе их вызова. Информация о виртуальных методах хранится в таблице виртуальных методов VMT. В VMT хранятся виртуальные методы данного класса и всех его предков. При создании потомка класса вся VMT предка переносится в потомок и там к ней добавляются новые методы. Поиск нужного метода занимает мало времени, так как класс имеет всю информацию о своих виртуальных методах. Динамические методы не дублируются в таблице динамических методов DMT потомка. DMT класса содержит только методы, объявленные в этом классе. При вызове динамического метода сначала осуществляется поиск в DMT данного класса, и если метод не найден – то в DMT предка класса и т.д. Таким образом использование виртуальных методов требует большего расхода памяти из-за необходимости хранения массивных VMT всех классов, зато они вызываются быстрее. Изменяя алгоритм того или иного метода в производных классах, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо перегрузить его в потомке, т. е. объявить в наследнике одноименный метод и реализовать в нем нужные действия. В результате в объекте родителе и объекте-потомке будут действовать два одноименных метода, имеющие разную алгоритмическую основу. Это называется полиморфизмом объектов. Виртуальные и динамические методы объявляются так же, как и статические, только в конце описания метода добавляются служебные слова virtual или dynamic:

type имя\_родителя =class ... метод; virtual; ... end;

Чтобы перегрузить в классе-наследнике виртуальный метод, нужно после его объявления написать ключевое слово override:

type имя\_наследника=class (имя\_родителя) ... метод; override; ... end;

Абстрактный метод – это виртуальный или динамический метод, реализация которого не определена в том классе, где он объявлен. Предполагается, что этот метод будет перегружен в классе-наследнике. Вызывают метод только в тех классах, где он перезагружен. Объявляется абстрактный метод при помощи служебного слова abstract после слов virtual или dynamic, например: метод1; virtual; abstract;

**Перегрузка операций**

Во Free Pascal можно перегрузить не только функции, но и операции, например, можно запрограммировать, чтобы операция \* при работе с матрицами осуществляла умножение матриц, а при работе с комплексными числами – умножение комплексных чисел. Для этого в программе нужно написать специальную функцию – метод. Объявление метода перегрузки записывается после объявления класса и выглядит так: operator symbols(параметры:тип)имя\_результата:тип; где operator – служебное слово; symbols – символ перегружаемой операции; параметры – имена переменных, участвующих в перегрузке оператора;

Конструкторы и деструкторы

Центральной идеей ООП является реализация понятия "абстракция".  
Смысл абстракции заключается в том, что сущность произвольной сложности  
можно рассматривать, а также производить определенные действия над ней, как над единым целым.  
Пример: Задача составления расписания занятий.  
Необходимые абстракции: студент, курс лекций, преподаватель, аудитория.  
Операции:  
− Определить студента в группу  
− Назначить аудиторию для группы  
− . . . . . . . . . .  
Одним из основных способов создания абстракции является использование  
концепции иерархической классификации. Ее суть заключается в том, что  
сложные системы разбиваются на более простые фрагменты.  
Практически все сложные системы иерархичны, и уровни их иерархии отражают различные уровни абстракции. Для каждой конкретной задачи рассматривается соответствующий уровень. Выбор низшего уровня абстракции достаточно произволен. Выбранный уровень в одном случае в качестве низшего уровня  
может оказаться уровнем достаточно высокой абстракции в другом проекте.  
Различают типовую иерархию и структурную иерархию, которые далее  
мы будем называть соответственно структурой классов и структурой объектов.

Конструкторы, возможно, не станут новинкой для тех, кто занимается структурным программированием. В некоторых объектно-ориентированных языках вроде Java и C# конструкторы представляют собой методы с тем же именем, что и соответствующий класс. В Visual Basic .NET используется обозначение New, а в Objective-C — ключевое слово init. Как обычно, мы сосредоточимся на концепциях конструкторов и не станем рассматривать специфический синтаксис всех языков. Взглянем на Java-код, реализующий конструктор. При создании нового объекта в первую очередь выполняется вызов конструктора. Пожалуй, наиболее важной функцией конструктора является инициализация выделенной памяти при обнаружении ключевого слова new. Коротко говоря, код, заключенный внутри конструктора, должен задать для нового созданного объекта его начальное, стабильное, надежное состояние. Функция, называемая служебной (или инициализацией), часто используется в целях инициализации при структурном программировании. Инициализация атрибутов пред­ставляет собой общую операцию, выполняемую в конструкторе. Если вы напишете класс и не добавите в него конструктор, то этот класс все равно пройдет компиляцию, и вы сможете его использовать. Если в классе не окажется предусмотрено явного конструктора, то будет обеспечен конструктор по умолчанию. Важно понимать, что в наличии всегда будет как минимум один конструктор вне зависимости от того, напишете ли вы его сами. Если вы не предусмотрите конструк­тора, то система обеспечит за вас конструктор по умолчанию.

Помимо создания объекта как такового, единственное действие, предприни­маемое конструктором по умолчанию, — это вызов конструктора его суперкласса. Если бы вам потребовалось декомпилировать байт-код, выдаваемый компиля­тором, то вы увидели бы этот код. Его в действительности вставляет компилятор.

В данном случае, если Cabbie не наследует явным образом от другого класса, то класс Object будет родительским. Пожалуй, в некоторых ситуациях конструктора по умолчанию окажется достаточно; однако в большинстве случаев потребуется инициализация памяти. Независимо от ситуации, правильная методика програм­мирования — всегда включать в класс минимум один конструктор. Если в классе содержатся атрибуты, то желательна их инициализация. Кроме того, инициализа­ция переменных всегда будет правильной методикой при написании кода, будь он объектно-ориентированным или нет.

**ОБЕСПЕЧЕНИЕ КОНСТРУКТОРА**

Общее правило заключается в том, что вы должны всегда обеспечивать конструктор, даже если не планируете что-либо делать внутри него. Вы можете предусмотреть кон­структор, в котором ничего нет, а затем добавить в него что-то. Хотя технически ничего плохого в использовании конструктора по умолчанию, обеспечиваемого компилятором, нет, в целях документирования и сопровождения никогда не будет лишним знать, как именно выглядит ваш код.

Неудивительно, что сопровождение становится здесь проблемой. Если вы за­висите от конструктора по умолчанию, а при последующем сопровождении будет добавлен еще один конструктор, то конструктор по умолчанию больше не будет обеспечиваться. Коротко говоря, конструктор по умолчанию добавляется, только если вы сами не включите никаких конструкторов. Как только вы предусмотрите хотя бы один конструктор, конструктор по умолчанию больше не будет обеспечи­ваться. **Использование множественных конструкторов**

Во многих случаях объект можно будет сконструировать несколькими метода­ми. Чтобы приспособиться к таким ситуациям, вам потребуется предусмотреть более одного конструктора. С одной стороны, мы хотим инициализировать атрибут count для отсчета до нуля — мы можем легко сделать это, использовав конструктор для инициализации count значением 0, как показано далее:

public Count(){

count = 0;

}

С другой стороны, нам может потребоваться передать параметр инициализации, который позволит задавать для count различные числовые значения:

public Count (int number){

count = number;

}

Это называется *перегрузкой метода* (перегрузка имеет отношение ко всем ме­тодам, а не только к конструкторам). В большинстве объектно-ориентированных языков предусматривается функциональность для перегрузки методов.

**Проектирование конструкторов**

Как вы уже видели ранее, при проектировании класса желательна инициализация всех атрибутов. В отдельных языках программирования компилятор обеспечивает некоторую инициализацию. Но, как и всегда, не следует рассчитывать на компилятор. в плане инициализации атрибутов! При использовании Java вы не сможете задей­ствовать тот или иной атрибут до тех пор, пока он не будет инициализирован. Если атрибут впервые задается в коде, то позаботьтесь о том, чтобы инициализировать его с каким-нибудь допустимым условием — например, определить для целочисленной переменной значение 0.

Конструкторы используются для обеспечения того, что приложения будут пребывать в стабильном состоянии (мне нравится называть его «надежным» со­стоянием). Например, инициализируя атрибут значением 0, можно получить не­стабильное приложение, если этот атрибут предназначается для использования в качестве делителя в операции деления. Вы должны учитывать, что деление на нуль — недопустимая операция. Инициализация значением 0 не всегда оказыва­ется наилучшим вариантом.

При проектировании правильная методика заключается в том, чтобы опреде­лить стабильное состояние для всех атрибутов, а затем инициализировать их с этим стабильным состоянием в конструкторе.

**Обработка ошибок**

Крайне редко бывает так, что тот или иной класс оказывается идеально написанным с первого раза. В большинстве, если не во всех ситуациях, *будут* ошибки. Любой раз­работчик, не имеющий плана действий на случай возникновения проблем, рискует.

Если ваш код способен выявлять и перехватывать ошибочные условия, то вы можете обрабатывать ошибки несколькими путями: в книге «Учебник по Java для начинающих» *(Java Primer Plus)* Пол Тима (Paul Tyma), Габриэл Торок (Gabriel Torok) и Трой Даунинг (Troy Downing) утверждают, что существует три основных подхода к проблемам, выявляемым в программах: устранить проблемы, игнориро­вать проблемы, отбросив их, или выйти из среды выполнения неким корректным образом. В книге «Объектно-ориентированное проектирование на Java» *(Object- Oriented Design in Java)* Гилберт и Маккарти более подробно останавливаются на этой теме, добавляя такой вариант, как возможность выбрасывать исключения.

 Игнорирование проблем — плохая идея!

 Проверка на предмет проблем и прерывание выполнения программы при их обнаружении.

 Проверка на предмет потенциальных проблем, перехват ошибок и попытка решить обнаруженные проблемы.

 Выбрасывание исключений (оно зачастую оказывается предпочтительным способом урегулирования соответствующих ситуаций).

Эти стратегии рассматриваются в приведенных далее разделах.

**Выбрасывание исключений**

В большинстве объектно-ориентированных языков программирования предусма­тривается такая функция, как *исключения*. В самом общем смысле под исключе­ниями понимаются неожиданные события, которые имеют место в системе. Ис­ключения дают возможность выявлять проблемы, а затем решать их. В Java, C#, C++, Objective-C и Visual Basic исключения обрабатываются при использовании ключевых слов catch и throw. Это может показаться игрой в бейсбол, однако ключе­вая концепция в данном случае заключается в том, что пишется определенный блок кода для обработки определенного исключения. Такая методика позволяет выяснить, где проблема берет свое начало, и раскрутить код до соответствующей точки. При выбрасывании исключения в блоке try оно будет обработано блоком catch. Если выбрасывание исключения произойдет, когда блок будет выполняться, то случится следующее.

1. Выполнение блока try завершится.

2. Предложения catch будут проверены с целью выяснить, надлежащий ли блок catch был включен для обработки проблемного исключения (на каждый блок try может приходиться более одного предложения catch).

3. Если ни одно из предложений catch не обработает проблемное исключение, то оно будет передано следующему блоку try более высокого уровня (если исклю­чение не будет перехвачено в коде, то система в конечном счете сама перехватит его, а результат будет непредсказуемым, то есть случится аварийное завершение приложения).

4. Если будет выявлено соответствующее предложение catch (обнаружено первое из соответствующих), то будут выполнены операторы в предложении catch.

5. Выполнение возобновится с оператора, следующего за блоком try.

Достаточно сказать, что исключения — серьезное преимущество объектно- ориентированных языков программирования. Вот пример того, как исключение перехватывается при использовании Java:

Как и при проектировании конструкторов, жизненно важно продумать, как класс будет обрабатывать ошибки. Обработка ошибок была подробно рассмотрена в главе 3.

Почти наверняка можно сказать, что каждая система будет сталкиваться с не­предвиденными проблемами. Поэтому не стоит игнорировать потенциальные ошибки. Разработчик хорошего класса (или любого кода, если на то пошло) пред­видит потенциальные ошибки и предусматривает код для обработки таких ситуа­ций, когда они имеют место.

Согласно общему правилу приложение никогда не должно завершаться аварий­но. При обнаружении ошибки система должна либо «починить» себя и продолжить функционировать, либо корректно завершить свою работу без потери каких-либо данных, важных для пользователя.

//////////////////////////////

При проектировании класса одна из самых важных соответствующих задач — при­нять решение о том, как этот класс будет сконструирован. Конструкторы были рассмотрены в главе 3. Загляните в нее снова, если вам потребуется освежить свои знания насчет основных принципов проектирования конструкторов.

Прежде всего конструктор должен задать для объекта его начальное, надежное состояние. Сюда входит выполнение таких задач, как инициализация атрибутов и управление памятью. Вам также потребуется убедиться в том, что объект долж­ным образом сконструирован в состоянии по умолчанию. Как вариант, можно обеспечить конструктор для обработки этой стандартной ситуации. При использовании языков программирования, в которых имеются деструкторы, жизненно важно, чтобы эти деструкторы включали соответствующие функции очист­ки. В большинстве случаев такая очистка связана с высвобождением системной памя­ти, полученной объектом в какой-то момент. Java и .NET автоматически регенерируют память с помощью механизма сборки мусора. При использовании языков программи­рования вроде C++ разработчик должен включать код в деструктор для надлежащего высвобождения памяти, которую объект занимал во время своего существования. Если проигнорировать эту функцию, то в результате произойдет утечка памяти.

Generics

http://www.freepascal.ru/article/freepascal/20070628121838/

Свойства

Свойства - это специальный механизм классов, регулирующий доступ к полям. Свойства объявляются с помощью зарезервированных слов property, read и write. Обычно свойство связано с некоторым полем и указывает те методы класса, которые должны использоваться при записи в это поле или при чтении из него. Синтаксис объявления свойств следующий:propety имя\_1: тип read имя\_чтения write имя\_2. Зарезервированное слово read описывает метод чтения свойств объекта, а слово write описывает метод записи свойств объекта.Имя\_1 и имя\_2 – соответственно имена методов, обеспечивающих чтение или запись свойства. Если необходимо, чтобы свойство было доступно только для чтения или только для записи, следует опустить соответственно часть write или read. Рассмотрим следующий пример. Создадим класс – многоугольник, имя класса TPolygon. Полями класса будут: • K – количество сторон многоугольника; • p – массив, в котором будут храниться длины сторон многоугольника. Методами класса будут: • конструктор Create, обнуляющий элементы массива p; • Perimetr() - функция вычисления периметра фигуры;• Show() - функция формирования сведений о фигуре (количество сторон и периметр); • Set\_Input() - функция проверки исходных данных.

Литература

* Расолько, Г. А. Сборнік задач па курсу «Метады праграміравання і информатика»: дапаможнік для студэнтаў мех. -мат. фак. спец. 1-31 03 01-02 «Матэматыка (навукова-педагагічная дзейнасць)». У 2 ч. Ч. ІI. Алгарытмы апрацоўкі даных / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень, Минск 2013г. – 91с.
* Расолько, Г. А. Метады праграміравання і інфарматыка: канспект лекцый для студэнтаў мех. -мат. фак. спец. 1-31 03 01-02 «Матэматыка (навукова-педагагічная дзейнасць)». У 2 ч. Ч. ІІ. Алгарытмы апрацоўкі даных / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень, Минск 2013г. – 100с.