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):

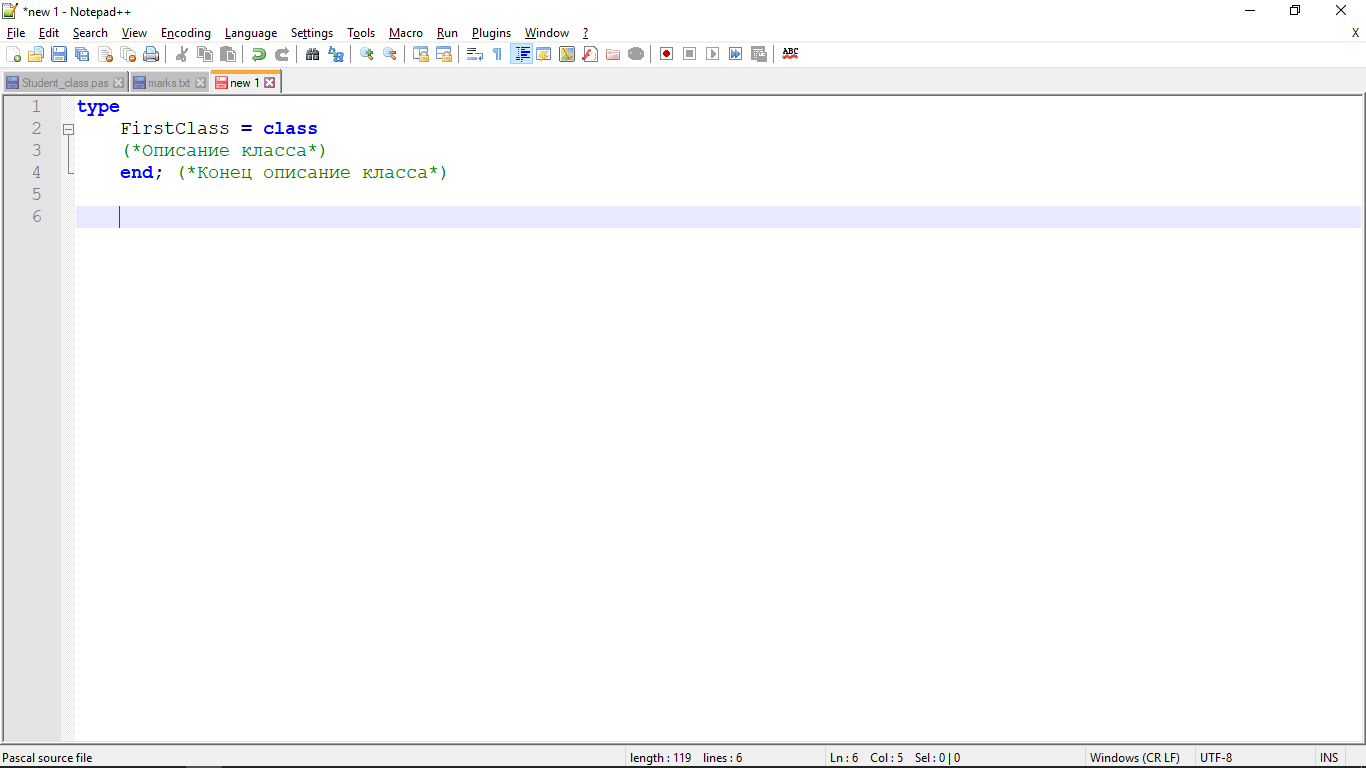


Рисунок 1

Атрибуты описываются как обычные переменные (Рисунок 2):

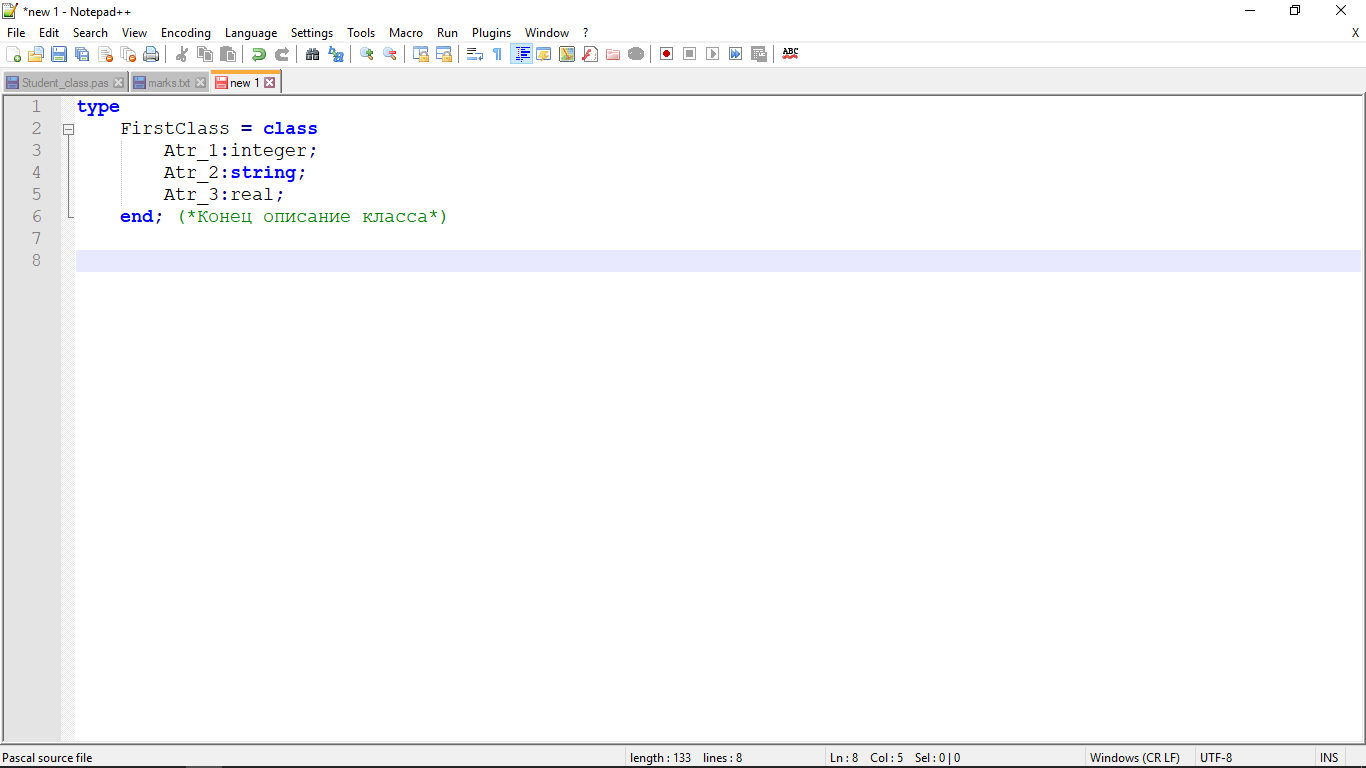


Рисунок 2

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

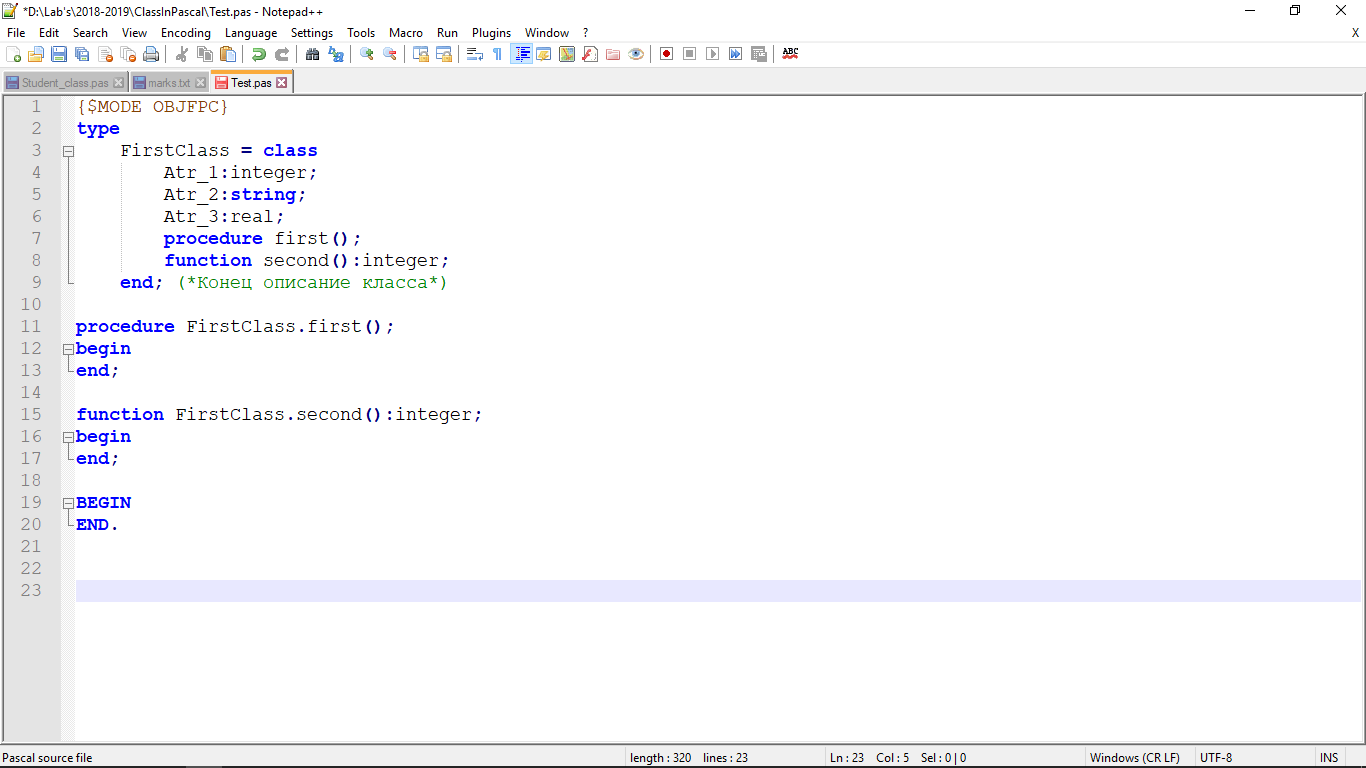


Рисунок 3

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.

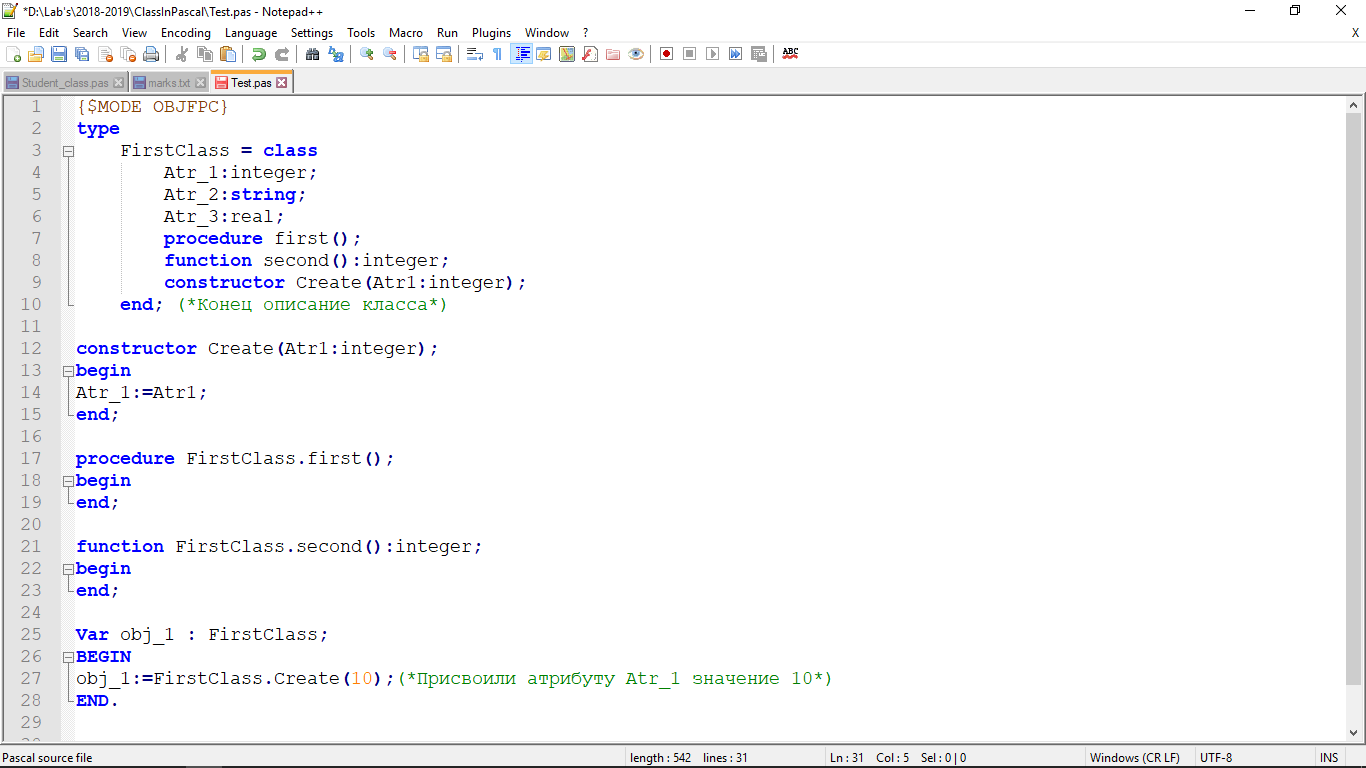


Рисунок 4

**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

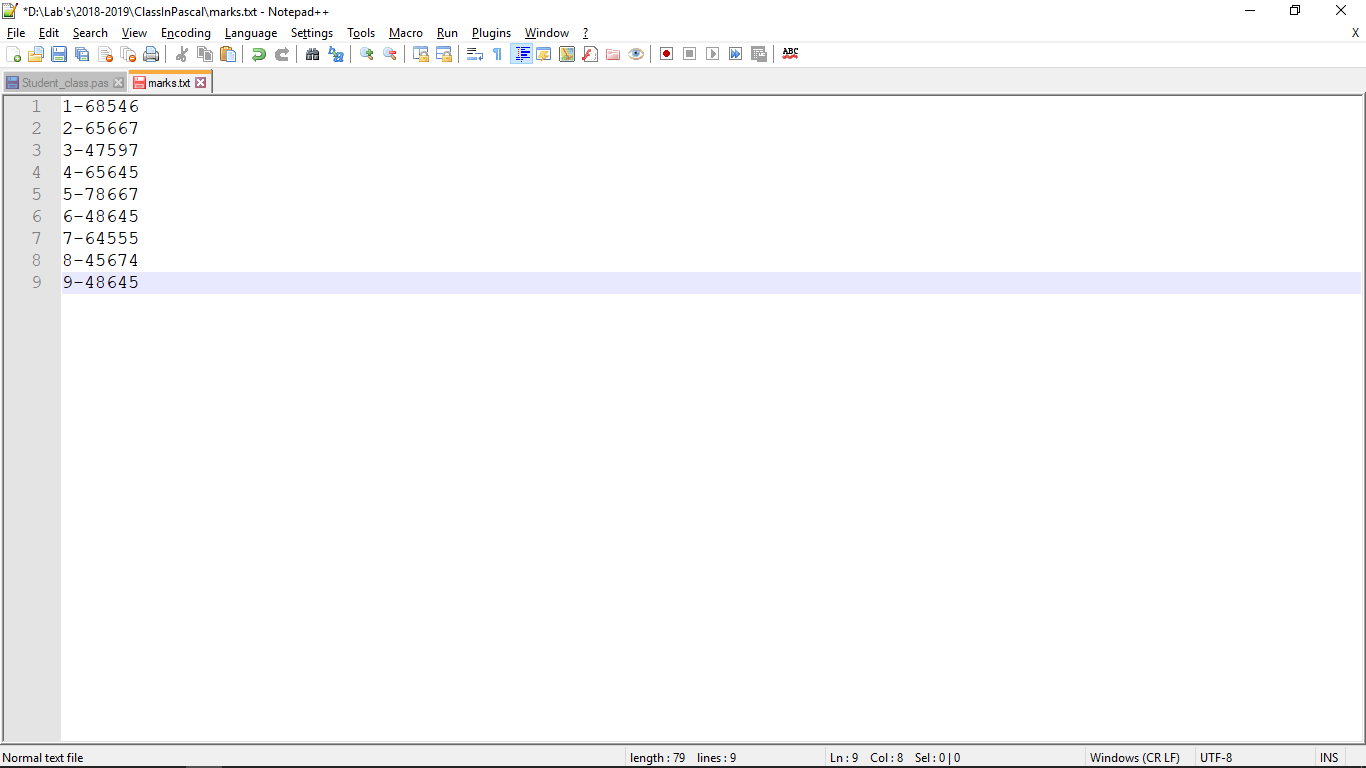


Рисунок 5

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

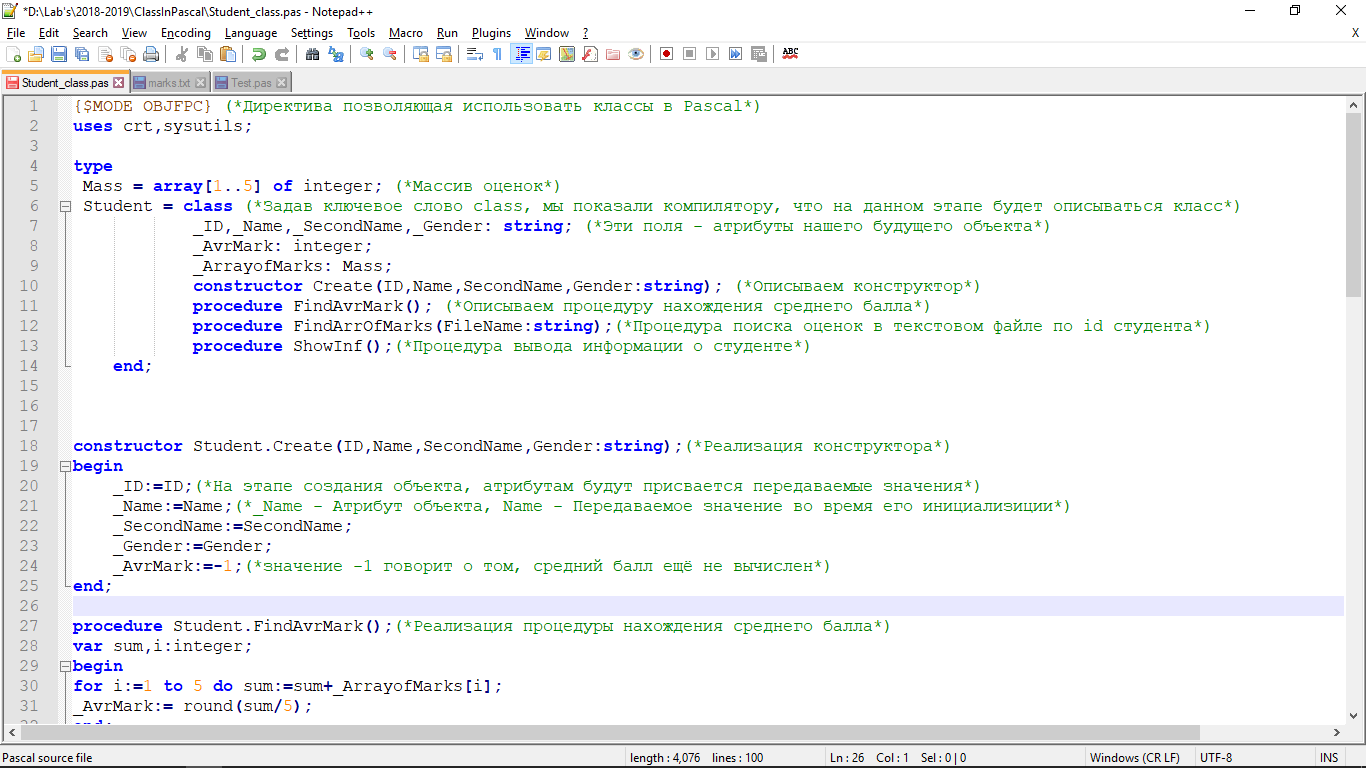


Рисунок 6

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):

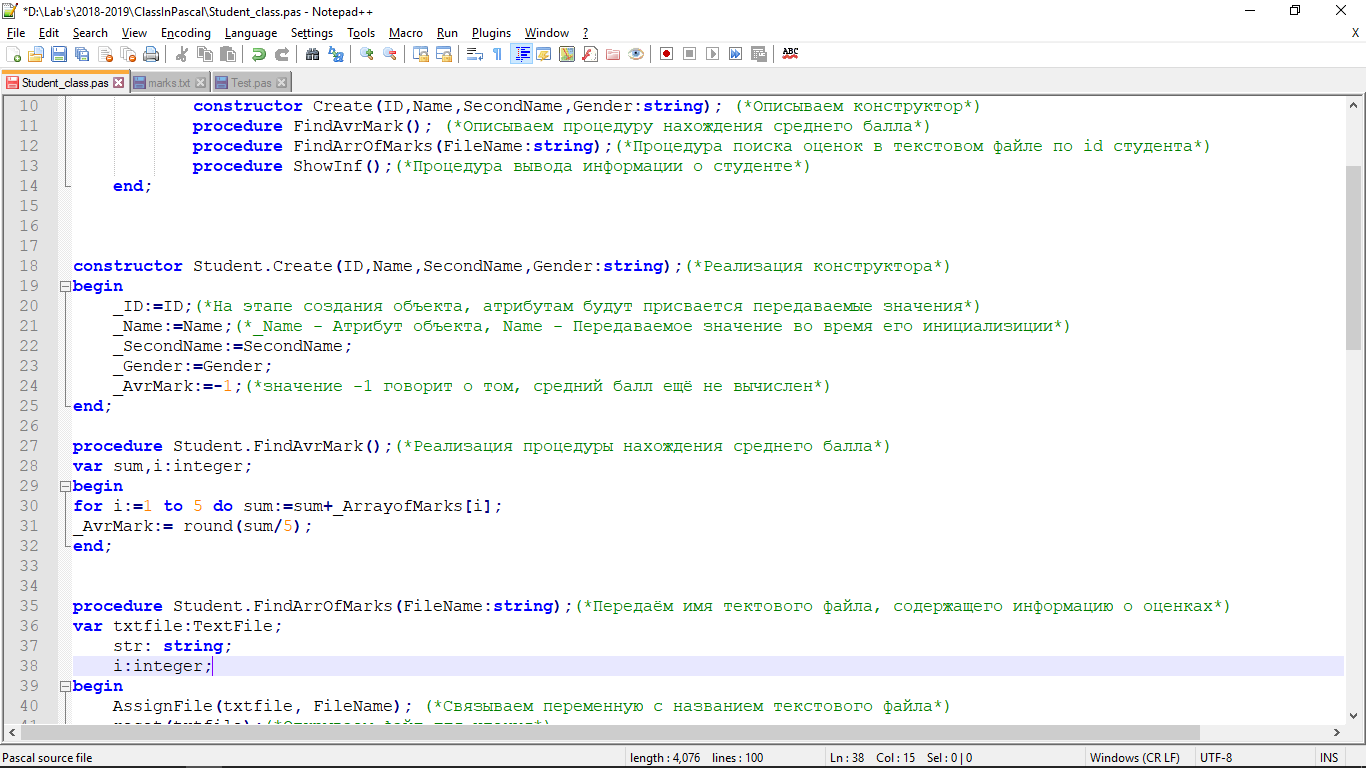


Рисунок 7

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):

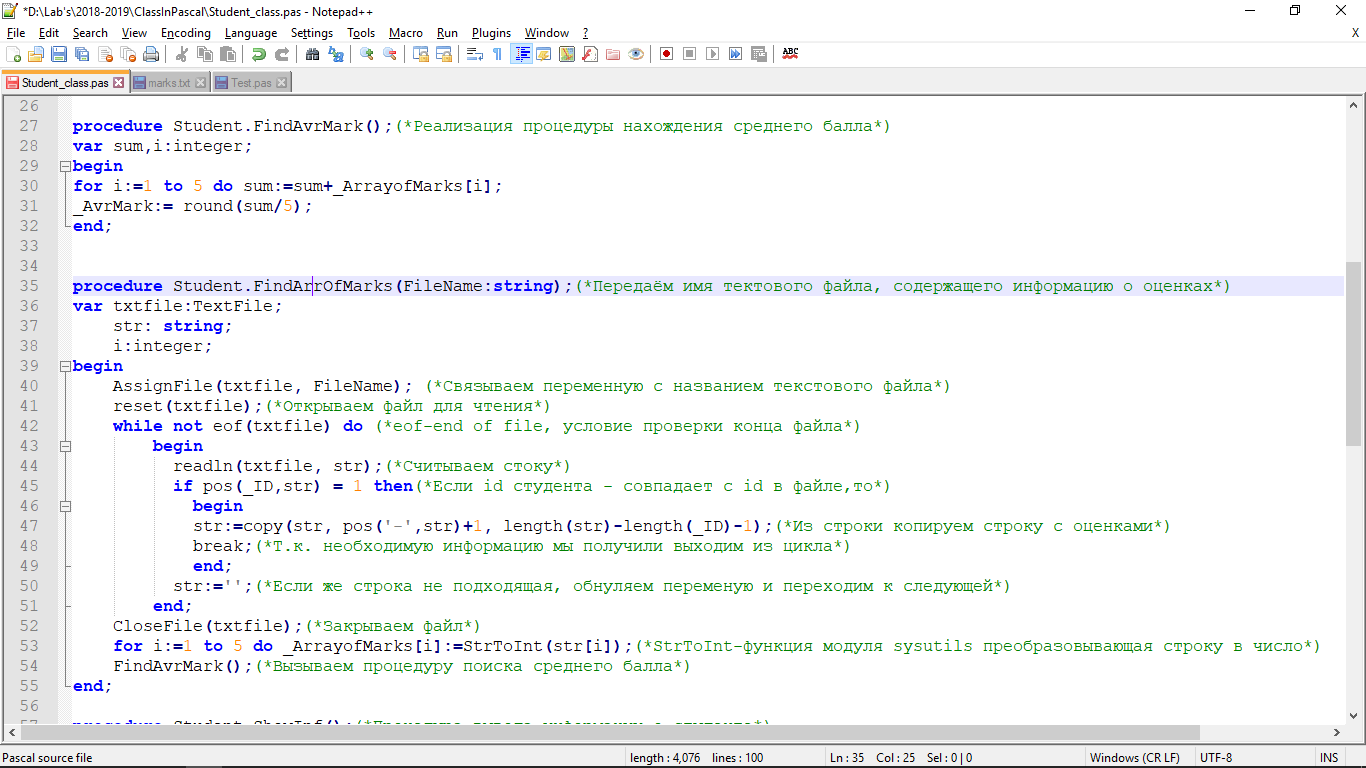


Рисунок 8

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

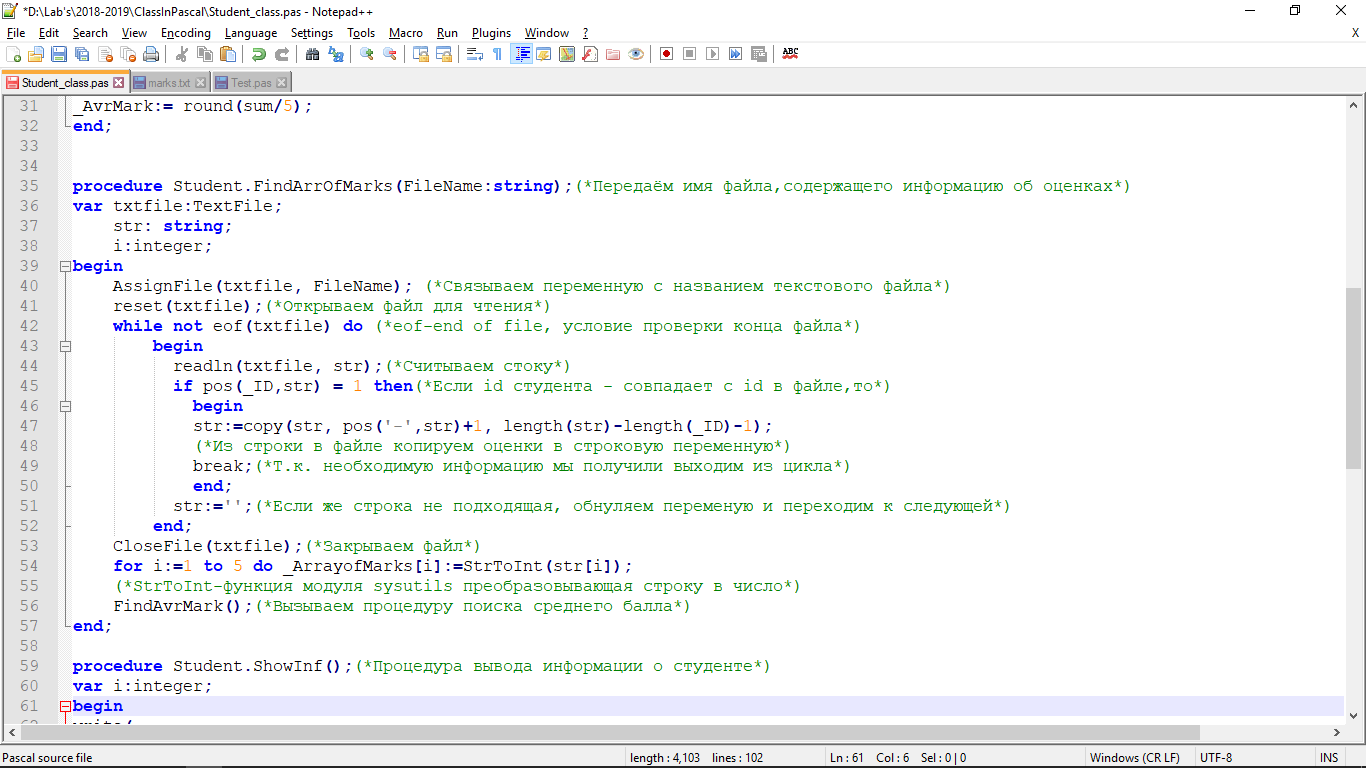


Рисунок 9

Процедура вывода информации о студенте (Рис.10):

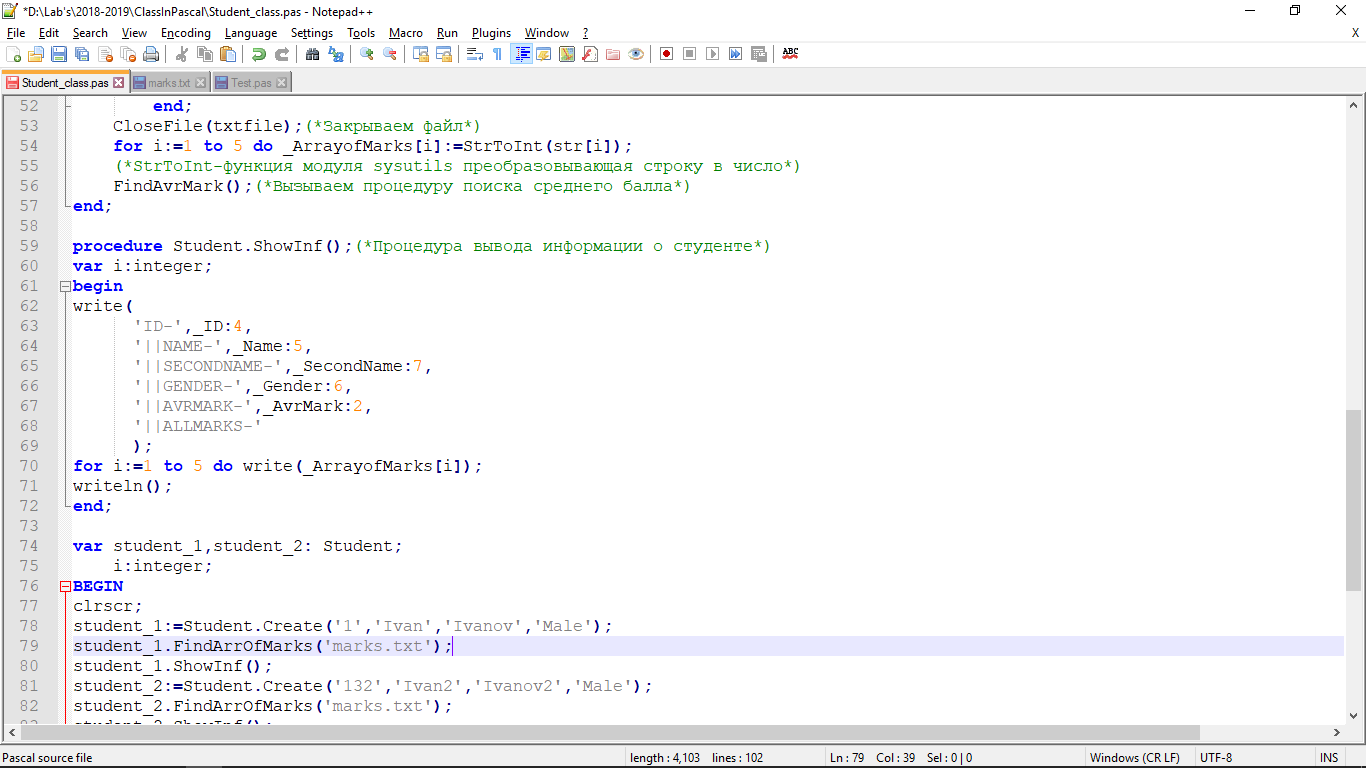


Рисунок 10

Код основной программы (Рис.11):

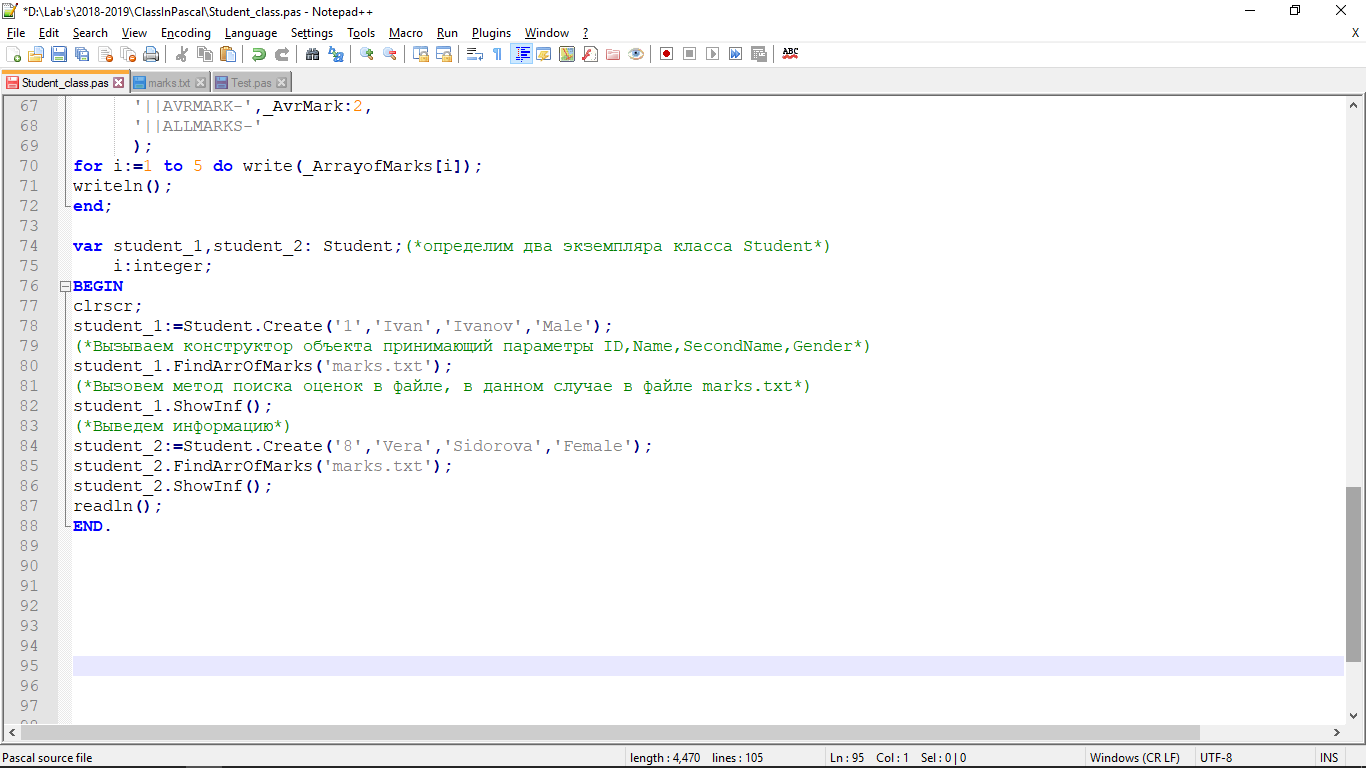
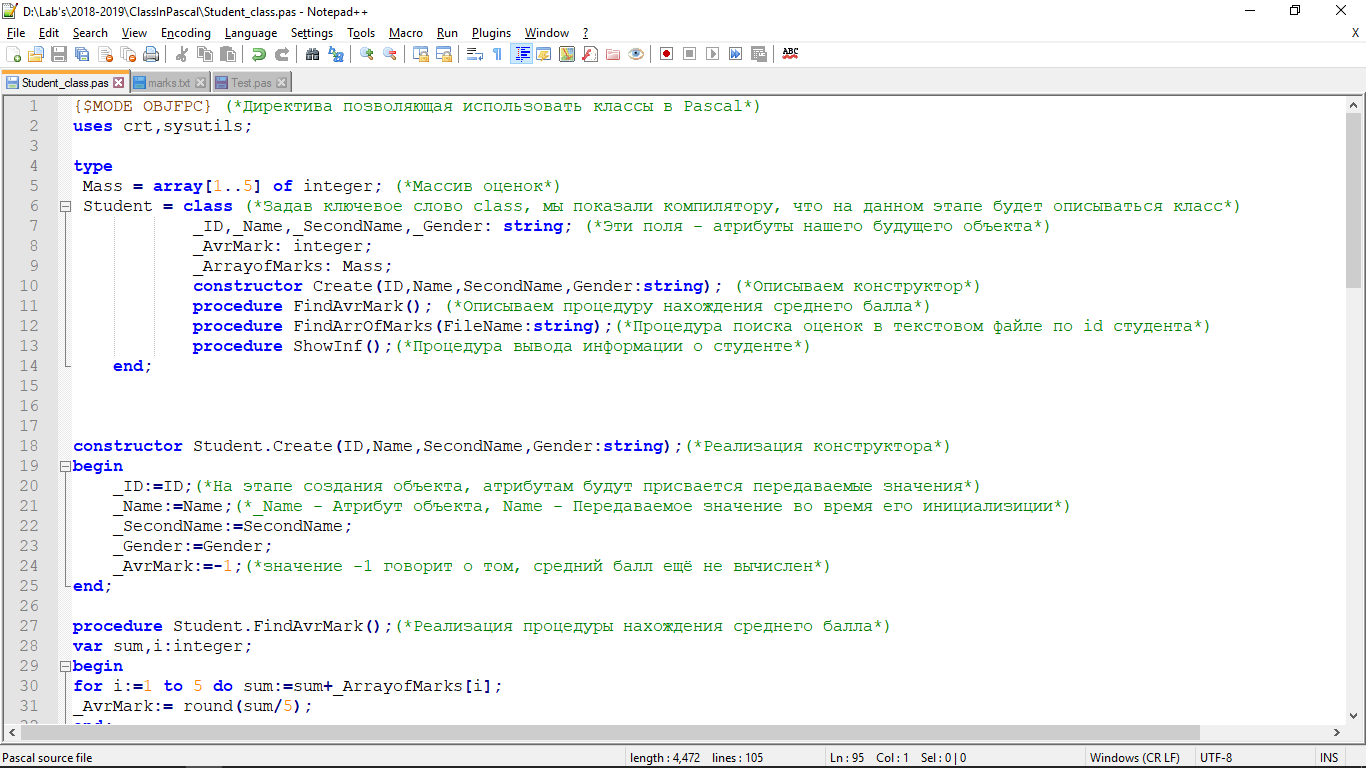
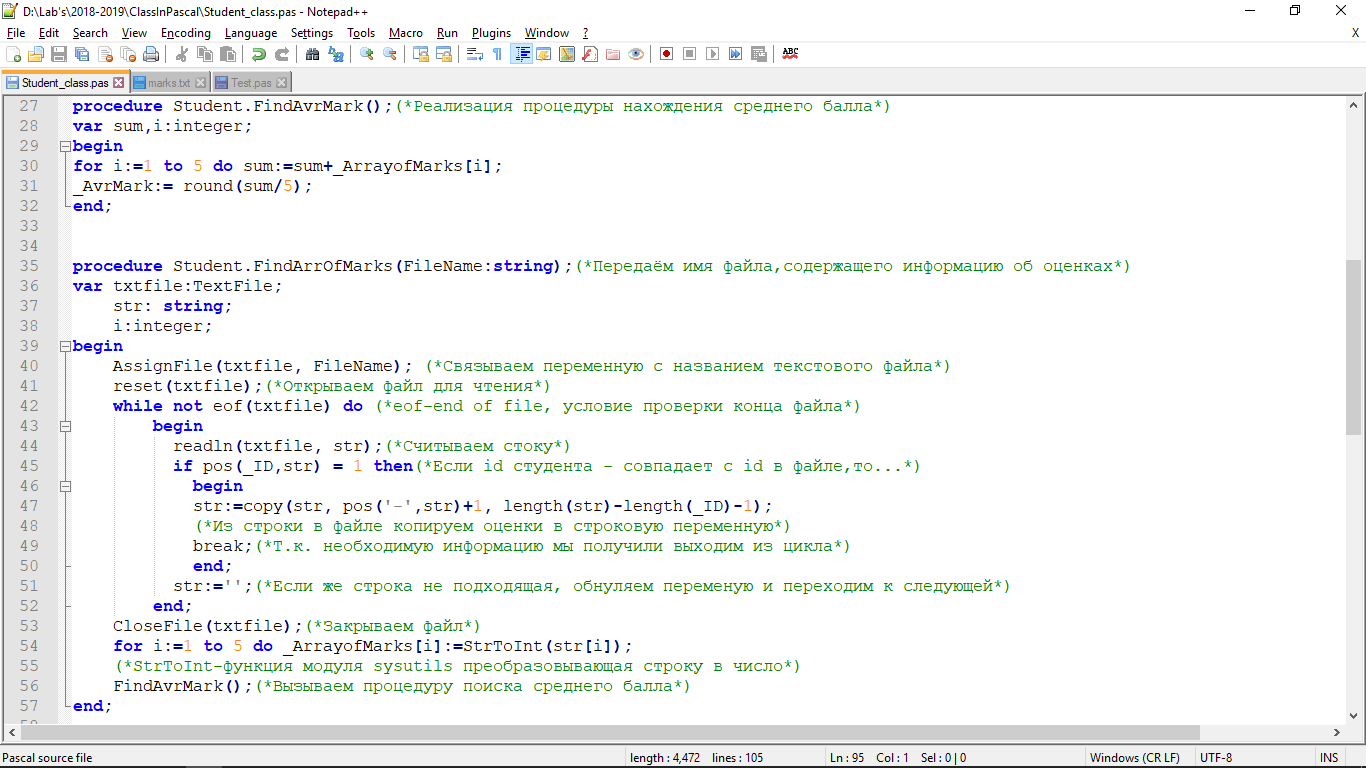


Рисунок 11

Весь код программы (Рис.12):





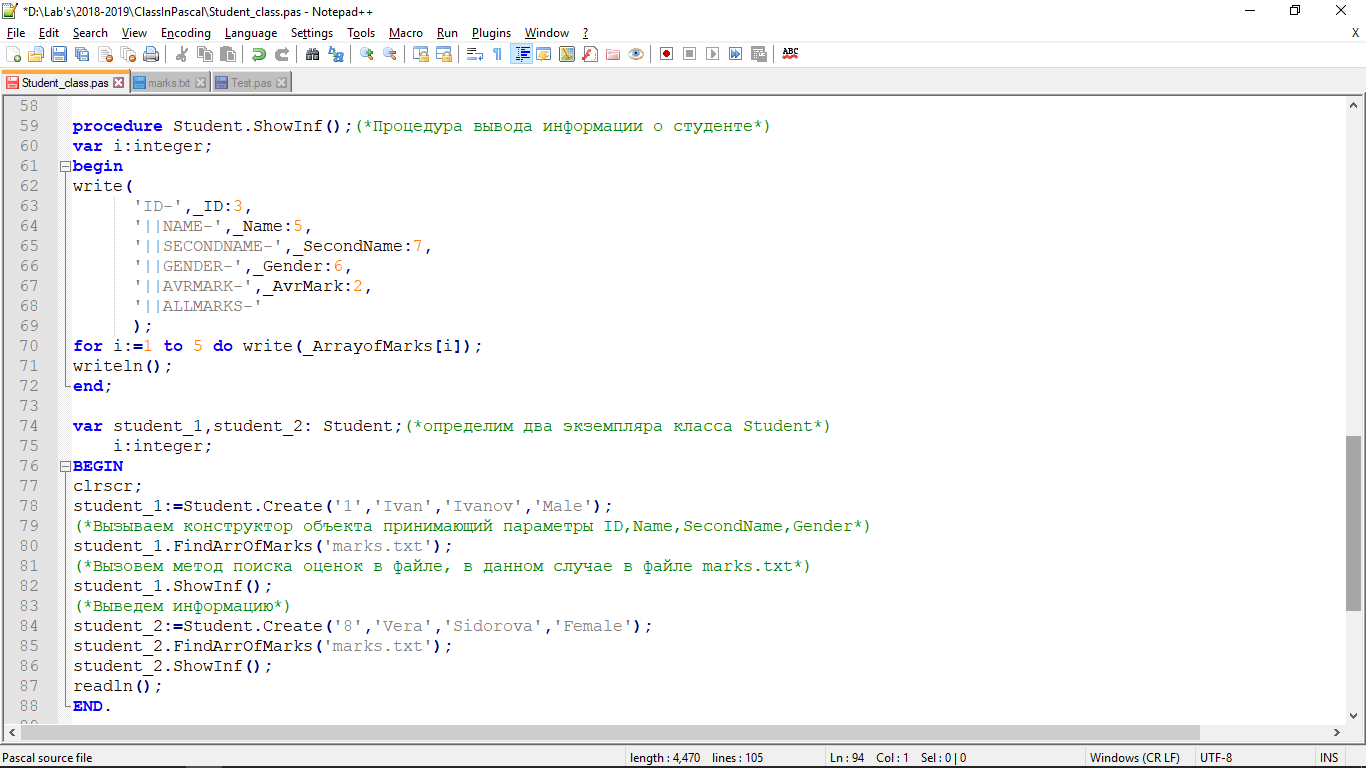


Рисунок 12

Результат выполнения программы (Рис.13):

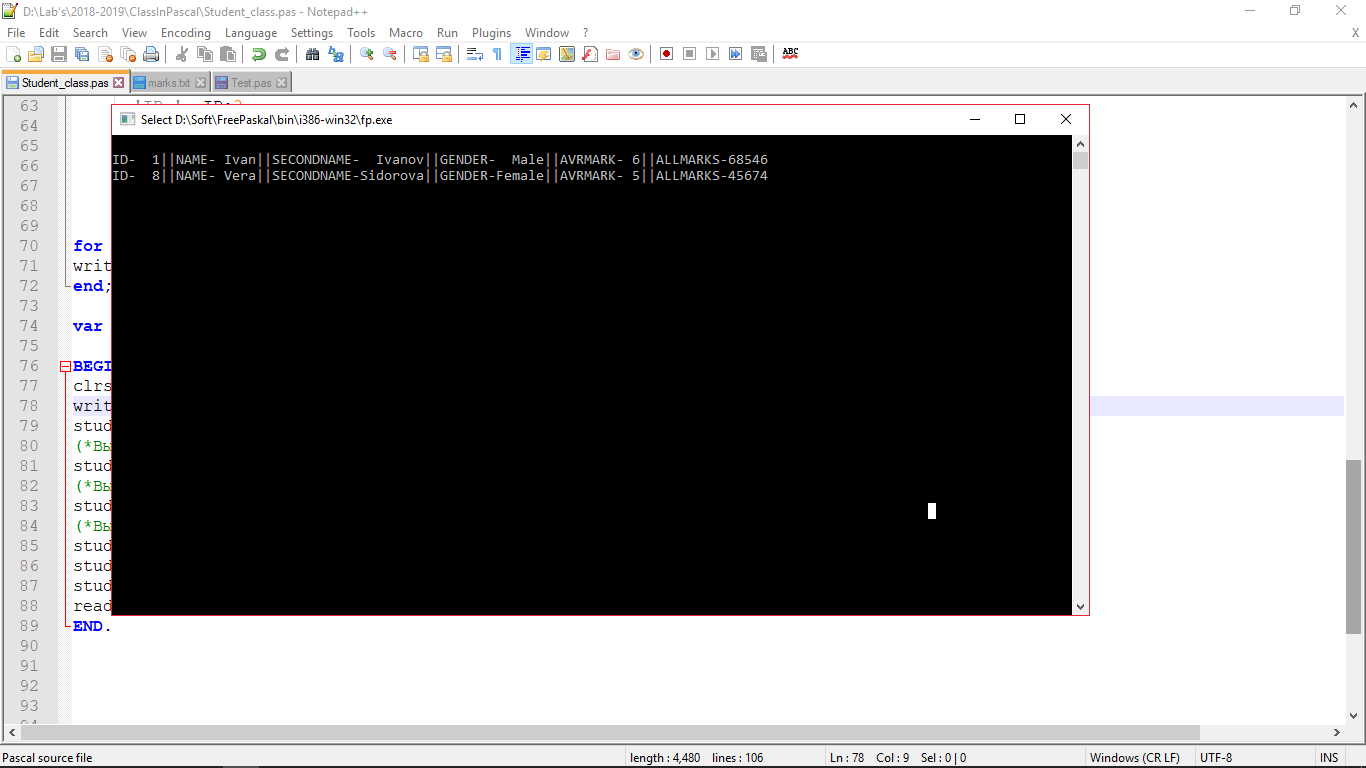


Рисунок 13

Абстрактные типы данных

Хотя термины тип данных (или просто тип), структура данных и абстрактный тип данных звучат похоже, но имеют они различный смысл. В языках программирования тип данных переменной обозначает множество значений, которые может принимать эта переменная. Например, переменная булевого (логического) типа может принимать только два значения: значение true (истина) и значениеfalse (ложь) и никакие другие. Набор базовых типов данных отличается в различных языках: в языкеPascal это типы целых (integer) и действительных (real) чисел, булев (boolean) тип и символьный (char) тип. Правила конструирования составных типов данных на основе базовых типов также различаются в разных языках программирования.Втаких языках высокого уровня, как Си и Pascal легко и быстро строить составных типы.Абстрактный тип данных (АТД) – это математическая модель плюс различные операторы, определенные в рамках этой модели. Алгоритм может разрабатываться в терминах АТД, но для реализации алгоритма в конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов, поддерживаемых данным языком программирования. Для представления АТД используются структуры данных, которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом.Базовым строительным блоком структуры данных является ячейка, которая предназначена для хранения значения определенного базового или составного типа данных. Структуры данных создаются путем задания имен совокупностям (агрегатам) ячеек и (необязательно) интерпретации значения некоторых ячеек как представителей (т. е. указателей) других ячеек.Вкачестве простейшего механизма агрегирования ячеек в Pascal и большинстве других языков программирования можно применять (одномерный) массив, т. е. последовательность ячеек определенного типа. Массив также можно рассматривать как отображение множества индексов (таких как целые числа 1, 2, …,п) во множество ячеек. Ссылка на ячейку обычно состоит из имени массива и значения из множества индексов данного массива. В языке Pascal множество индексов может быть нечислового типа, например(север, восток, юг, запад), или интервального типа (как 1..10). Значения всех ячеек массива должны иметь одинаковый тип данных. Объявление имя: array[ТипИндекса]of ТипЯчеек;задает имя для последовательности ячеек, тип для элементов множества индексов и тип содержимого ячеек. Кстати, Pascal необычайно богат на типы индексов. Многие языки программирования позволяют использовать в качестве индексов только множества последовательных целых чисел. Например, чтобы в языкеFortran Алгоритмы и структуры данных. Пособие по самостоятельной работе131. ОБЩИЕ СВЕДЕНИЯ ОБ АЛГОРИТМАХ1.3.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.Другим общим механизмом агрегирования ячеек в языках программирования является структура записи. Запись (record) можно рассматривать как ячейку, состоящую из нескольких других ячеек (называемых полями), значения в которых могут быть разных типов. Записи часто группируются в массивы; тип данных определяется совокупностью типов полей записи. Например, вPascal объявлениеvarreclist: array[1..4] of record data: real;next: integerendзадает имя reclist (список записей)4-элементногомассива, значениями которого являются записи с двумя полями:data (данные) иnext (следующий).Третий метод агрегирования ячеек, который можно найти в Pascal и некоторых других языках программирования, – это файл. Файл, как и одномерный массив, является последовательностью значений определенного типа. Однако файл не имеет индексов: его элементы доступны только в том порядке, в каком они были записаны в файл. В отличие от файла, массивы и записи являются структурами с «произвольным доступом», подразумевая под этим, что время доступа к компонентам массива или записи не зависит от значения индекса массива или указателя поля записи. Достоинство агрегирования с помощью файла (частично компенсирующее описанный недостаток) заключается в том, что файл не имеет ограничения на количество составляющих его элементов и это количество может изменяться во время выпол - нения программы.1.4. АбстрактныетипыданныхПеред тем, как рассмотреть абстрактный тип данных, обсудим его роль в процессе разработки программ. Сначала сравнивним абстрактный тип данных с таким знакомым понятием, как процедура.Процедуру, неотъемлемый инструмент программирования, можно рассматривать как обобщенное понятие оператора. В отличие от ограниченных по своим возможностям встроенных операторов языка программирования (сложения, умножения и т. п.), с помощью процедур программист может создавать собственные операторы и применять их к операндам различных типов, не только базовым. Примером такой процедуры-оператораможет служить стандартная подпрограмма перемножения матриц. Алгоритмы и структуры данных. Пособие по самостоятельной работе141. ОБЩИЕ СВЕДЕНИЯ ОБ АЛГОРИТМАХ1.4. Абстрактные типы данныхДругим преимуществом процедур (кроме способности создавать новые операторы) является возможность использования их для инкапсулирования частей алгоритма путем помещения в отдельный раздел программы всех операторов, отвечающих за определенный аспект функционирования программы. Пример инкапсуляции: использование одной процедуры для чтения входных данных любого типа и проверки их корректности. Преимущество инкапсуляции заключается в том, что мы знаем, какие инкапсулированные операторы необходимо изменить в случае возникновения проблем в функционировании программы. Например, если необходимо организовать проверку, являются ли значения входных данных положительными, следует изменить только несколько строк кода, и мы точно знаем, где эти строки находятся.ОпределениеабстрактноготипаданныхОпределим абстрактный тип данных (АТД) как математическую модель с совокупностью операторов, определенных в рамках этой модели. Простым примером АТД могут служить множества целых чисел с операторами объединения, пересечения и разности множеств. В модели АТД операторы могут иметь операндами не только данные, определенные АТД, но и данные других типов: стандартных типов языка программирования или определенных в других АТД. Результат действия оператора также может иметь тип, отличный от определенных в данной модели АТД. Но мы предполагаем, что по крайней мене один операнд или результат любого оператора имеет тип данных, определенный в рассматриваемой модели абстрактных типов данных.Две характерные особенности процедур – обобщение и инкапсуляция, – о которых говорилось выше, отлично характеризуют абстрактный тип данных. АТД можно рассматривать как обобщение простых типов данных (целых и действительных чисел и т. д.), точно так же, как процедура является обобщением простых операторов (+, – и т. д.). Абстрактный тип данных инкапсулирует типы данных в том смысле, что определение типа и все операторы, выполняемые над данными этого типа, помещаются в один раздел программы. Если необходимо изменить реализацию АТД, мы знаем, где найти и что изменить в одном небольшом разделе программы, и можем быть уверенными, что это не приведет к ошибкам где-либов программе при работе с этим типом данных. Более того, вне раздела с определением операторов АТД можно рассматривать типы АТД как первичные типы, так как объявление типов формально не связано с их реализацией. Однако в этом случае могут возникнуть сложности, так как некоторые операторы могут инициироваться для более одного АТД и ссылки на эти операторы должны быть в разделах нескольких АТД.Можно реализовать тип данных любым способом, а программы, использующие объекты этого типа, не зависят от способа реализации типа – за это отвечают процедуры, реализующие операторы для этого типа данных. Алгоритмы и структуры данных. Пособие по самостоятельной работе151. ОБЩИЕ СВЕДЕНИЯ ОБ АЛГОРИТМАХ1.4. Абстрактные типы данныхТермин реализация АТД подразумевает следующее: перевод в операторы языка программирования объявлений, определяющие переменные этого абстрактного типа данных, плюс процедуры для каждого оператора, выполняемого над объектами АТД. Реализация зависит от структуры данных, представляющих АТД. 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 в языке Pascal. Например, одной из возможных реализации переменнойS типаSET может служить массив, содержащий элементы множества S.Одной из основных причин определения двух различных АТД в рамках одной модели является то, что над объектами этих АТД необходимо выполнять различные действия, т. е. определять операторы разных типов.В идеале желательно писать программы на языке, базовых типов данных и операторов которого достаточно для реализации АТД. С этой точки зрения язык Pascal не очень подходящий язык для реализации различных АТД, но, с другой стороны, трудно найти иной язык программирования, в котором можно было бы так непосредственно декларировать абстрактный тип данных.

Инкапсуляция

Место для ввода текста.

Инкапсуляция (encapsulation) - это механизм, который объединяет данные и код, манипулирующий зтими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании код и данные могут быть объединены вместе; в этом случае говорят, что создаётся так называемый "чёрный ящик". Когда коды и данные объединяются таким способом, создаётся объект (object). Другими словами, объект - это то, что поддерживает инкапсуляцию.

Внутри объекта коды и данные могут быть закрытыми (private). Закрытые коды или данные доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта. Если коды и данные являются открытыми, то, несмотря на то, что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов 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.

Одно из основных преимуществ использования объектов заключается в том, что объекту не нужно показывать все свои атрибуты и поведения. При хорошем объектно- ориентированном проектировании (по крайней мере, при таком, которое повсемест­но считается хорошим) объект должен показывать только интерфейсы, необходимые другим объектам для взаимодействия с ним. Детали, не относящиеся к использова­нию объекта, 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

Например, объект, который применяется для вычисления квадратов чисел, должен обеспечивать интерфейс для получения результатов. Однако внутренние атрибуты и алгоритмы, используемые для вычисления квадратов чисел, не нужно делать доступными для запрашивающего объекта. Надежные классы проектиру­ются с учетом инкапсуляции. В последующих разделах мы рассмотрим концепции интерфейса и реализации, которые образуют основу инкапсуляции.

Инкапсуляция — фундаментальная объектно-ориентированная концепция. Каждый раз при рассмотрении парадигмы «интерфейс/реализация» мы говорим об инкап­суляции. Основной вопрос заключается в том, что в классе должно быть видно, а что — нет. Инкапсуляция в равной мере касается данных и поведений. Когда речь идет о классе, то первоочередное проектное решение «вращается» вокруг инкапсу­ляции как данных, так и поведений в хорошо написанном классе.

Гилберт и Маккарти определяют инкапсуляцию как «процесс упаковки вашей программы с разделением каждого из ее классов на две обособленные части — интер­фейс и реализацию». Эта идея многократно повторяется и по ходу нашей книги.

Но при чем здесь инкапсуляция и какое отношение она имеет к этой главе? В данном случае мы имеем дело с объектно-ориентированным парадоксом. Ин­капсуляция является настолько фундаментальной объектно-ориентированной концепцией, что представляет собой одно из главных правил ООП. Наследова­ние тоже считается одной из трех важнейших объектно-ориентированных кон­цепций. Однако оно некоторым образом фактически нарушает инкапсуляцию!

**Как наследование ослабляет инкапсуляцию**

Как уже говорилось, инкапсуляция — это процесс упаковки классов в открытый интерфейс и закрытую реализацию. По сути в классе скрывается все, о чем другим классам знать не обязательно.

Петер Коуд и Марк Мейфилд отмечают, что при использовании наследования инкапсуляция, в сущности, ослабляется в рамках иерархии классов. Они говорят о конкретном риске: наследование означает сильную инкапсуляцию по отношению к остальным классам, но слабую инкапсуляцию между суперклассом и его под­классами.

Проблема заключается в том, что если от суперкласса будет унаследована реа­лизация, которая затем подвергнется модификации, то такое изменение *распро­странится* по иерархии классов. Этот волновой эффект потенциально способен затронуть все подклассы. Поначалу это может не показаться большой проблемой, однако, как мы уже видели ранее, подобный волновой эффект может привести к непредвиденным проблемам. Например, тестирование превратится в кошмар. В главе 6 мы говорили о том, как инкапсуляция упрощает системы тестирования. В теории, если вы создадите класс с именем Cabbie (рис. 7.9) и соответствующими открытыми интерфейсами, любое изменение реализации Cabbie должно быть прозрачным для всех остальных классов. Однако в любой конструкции изменение суперкласса, безусловно, нельзя назвать прозрачным для того или иного подклас­са. Понимаете, в чем проблема?

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>

Наследование

Одной из наиболее сильных сторон объектно-ориентированного программирова­ния, пожалуй, является 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. Однако объектно-ориентированное проектирование делает важный шаг вперед, позволяя вам определять отношения между классами, которые не только облегча­ют повторное использование кода, но и способствуют созданию лучшей общей конструкции путем упорядочения и учета общности разнообразных классов. Основное средство обеспечения такой функциональности — *наследование*.

Наследование позволяет классу наследовать атрибуты и методы другого класса. Это дает возможность создавать абсолютно новые классы путем абстрагирования общих атрибутов и поведений.

Одна из основных задач проектирования при объектно-ориентированном про­граммировании заключается в выделении общности разнообразных классов. До­пустим, у вас есть класс Dog и класс Cat, каждый из которых будет содержать атри­бут eyeColor. При процедурной модели код как для Dog, так и для Cat включал бы этот атрибут. При объектно-ориентированном проектировании атрибут, связанный с цветом, можно перенести в класс с именем Mammal наряду со всеми прочими об­щими атрибутами и методами. В данном случае оба класса — Dog и Cat — будут наследовать от класса Mammal, как показано на рис. 1.14.

**Суперклассы и подклассы**

Суперкласс, или родительский класс (иногда называемый базовым), содержит все атрибуты и поведения, общие для классов, которые наследуют от него. Например, в случае с классом Mammal все классы млекопитающих содержат аналогичные атри­буты, такие как eyeColor и hairColor, а также поведения вроде generateInternalHeat и growHair. Все классы млекопитающих включают эти атрибуты и поведения, поэто­му нет необходимости дублировать их, спускаясь по дереву наследования, для каждого типа млекопитающих. Дублирование потребует много дополнительной работы, и, пожалуй, что вызывает наибольшее беспокойство, оно может привести к ошибкам и несоответствиям.

Подкласс, или дочерний класс (иногда называемый производным), представ­ляет собой расширение суперкласса. Таким образом, классы Dog и Cat наследуют все общие атрибуты и поведения от класса Mammal. Класс Mammal считается супер­классом подклассов, или дочерних классов, Dog и Cat.

Наследование обеспечивает большое количество преимуществ в плане проектиро­вания. При проектировании класса Cat класс Mammal предоставляет значительную часть требуемой функциональности. Наследуя от объекта Mammal, Cat уже содержит все атри­буты и поведения, которые делают его настоящим классом млекопитающих. Точнее говоря, являясь классом млекопитающих такого типа, как кошки, Cat должен включать любые атрибуты и поведения, которые свойственны исключительно кошкам.  
Дерево наследования может разрастись довольно сильно. Когда классы Mammal и Cat будут готовы, добавить другие классы млекопитающих, например собак (или львов, тигров и медведей), не составит особого труда. Класс Cat также может выступать в роли суперкласса. Например, может потребоваться дополнительно абстрагировать Cat, чтобы обеспечить классы для персидских, сиамских кошек и т. д. Точно так же, как и Cat, класс Dog может выступать в роли родительского класса для других клас­сов, например GermanShepherd и Poodle (рис. 1.15). Мощь наследования заключается в его методиках абстрагирования и организации.

В большинстве объектно-ориентированных языков программирования (напри­мер, Java, .NET и Objective C) у класса может иметься 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*.  
Рассмотрим пример, в котором Circle, Square и Star наследуют от Shape. Это от­ношение часто называется *отношением «является экземпляром»*, поскольку круг — это форма, как и квадрат. Когда подкласс наследует от суперкласса, он получает все возможности, которыми обладает этот суперкласс. Таким образом, Circle, Square и Star являются расширениями Shape.

На рис. 1.17 имя каждого из объектов представляет метод Draw для Circle, Star и Square соответственно. При проектировании системы Shape очень полезно было бы стандартизировать то, как мы используем разнообразные формы. Так мы могли бы решить, что, если нам потребуется нарисовать фигуру любой фор­мы, мы вызовем метод с именем Draw. Если мы станем придерживаться этого решения всякий раз, когда нам нужно будет нарисовать фигуру, то потребуется вызывать только метод Draw, независимо от того, какой она будет формы. В этом заключается фундаментальная концепция полиморфизма — на индивидуальный объект, будь то Circle, Star или Square, возлагается обязанность по рисованию фигуры, которая ему соответствует. Это общая концепция во многих современ­ных приложениях, например, предназначенных для рисования и обработки текста.

**Как сконструирован суперкласс**

При использовании наследования вы должны знать, как сконструирован соответ­ствующий родительский класс. Помните, что, когда оно задействуется, от роди­тельского класса наследуется все. Таким образом, вам потребуется очень хорошо знать все данные и поведения родительского класса. Наследование атрибутов до­вольно очевидно. Однако то, как наследуются конструкторы, не так очевидно. После обнаружения ключевого слова new и выделения памяти для объекта пред­принимаются следующие шаги (рис. 3.4).

1. Внутри конструктора происходит вызов конструктора суперкласса соответ­ствующего класса. Если явного вызова конструктора суперкласса нет, то авто­матически вызывается конструктор по умолчанию. При этом вы сможете увидеть соответствующий код, взглянув на байт-коды.

2. Инициализируется каждый атрибут класса объекта. Эти атрибуты являются частью определения класса (переменные экземпляра), а не атрибутами внутри конструктора или любого другого метода (локальные переменные). В коде DataBaseReader, показанном ранее, целочисленная переменная startPosition яв­ляется переменной экземпляра класса.

3. Выполняется остальная часть кода внутри конструктора.

**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.

Обратите внимание, что метод getCompanyName объявлен как static, как метод класса; методы классов подробнее рассматриваются в главе 3. Помните, что атрибут companyName тоже объявлен как static. Метод, как и атрибут, может быть объявлен как static для указания на то, что на весь соответствующий класс приходится только одна копия этого метода.

**ОБЪЕКТЫ**

Вообще говоря, на каждый объект не приходится по одной физической копии каждого нестатического метода. В этом случае каждый объект указывал бы на один и тот же ма­шинный код. Однако на концептуальном уровне вы можете представлять, что объекты полностью независимы и содержат собственные атрибуты и методы.

**Методы закрытых реализаций**

Несмотря на то что все методы, рассмотренные в этой главе, определяются как public, не все методы в классе являются частью открытого интерфейса. Методы в том или ином классе обычно скрыты от других классов и объявляются как private:

Эти закрытые методы призваны быть частью реализации, а не открытого интерфейса. У вас может возникнуть вопрос насчет того, кто будет вызывать данные методы, если этого не сможет сделать ни один другой класс. Ответ прост: вы, возможно, уже подозревали, что эти методы можно вызывать изнутри мето­да giveDestination:

В качестве еще одного примера можно привести возможную ситуацию, когда у вас имеется внутренний метод, обеспечивающий шифрование, который вы буде­те вызывать изнутри самого класса. Коротко говоря, этот метод шифрования нель­зя вызвать извне созданного экземпляра объекта как такового.

Главное здесь состоит в том, что закрытые методы являются строго частью реа­лизации и недоступны другим классам.

**Резюме**

В этой главе мы заглянули внутрь класса и рассмотрели фундаментальные концеп­ции, необходимые для понимания принципов создания классов. Хотя в этой главе был использован скорее практический подход, в главе 5 классы будут рассмотрены с общей точки зрения проектировщика.

Полиморфизм

*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.

Возьмем, к примеру, класс Shape и поведение с именем Draw. Когда вы попро­сите кого-то нарисовать фигуру, первый вопрос вам будет звучать так: «Какой формы?» Никто не сможет нарисовать требуемую фигуру, не зная формы, которая является абстрактной концепцией (кстати, метод Draw() в коде Shape не содержит реализации). Вы должны указать конкретную форму. Для этого потребуется обе­спечить фактическую реализацию в Circle. Несмотря на то что Shape содержит метод Draw, Circle переопределит этот метод и обеспечит собственный метод Draw(). Переопределение, в сущности, означает замену реализации родительского класса на реализацию из дочернего класса.

Допустим, у вас имеется массив из трех форм — Circle, Square и Star. Даже если вы будете рассматривать их все как объекты Shape и отправите сообщение Draw каждому объекту Shape, то конечный результат для каждого из них будет разным, поскольку Circle, Square и Star обеспечивают фактические реализации. Одним словом, каждый класс способен реагировать на один и тот же метод Draw не так, как другие, и рисовать соответствующую фигуру. Это и понимается под полимор­физмом.

Здесь мы познакомимся с новой концепцией под названием *«конструктор»*. Класс Circle содержит метод с таким же именем — Circle. Если имя метода оказывается аналогичным имени класса и при этом не предусматривается возвращаемого типа, то это особый метод, называемый конструктором. Считайте конструктор точкой входа для класса, где создается объект. Конструктор хорошо подходит для выпол­нения инициализаций и задач, связанных с запуском.

Конструктор Circle принимает один параметр, представляющий радиус, и при­сваивает его атрибуту radius класса Circle.

Класс Circle также обеспечивает реализацию для метода getArea, изначально определенного как abstract в классе Shape.

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.

**Перегрузка методов**

Перегрузка позволяет программистам снова и снова использовать один и тот же метод, если его подпись каждый раз отличается. Подпись состоит из имени метода и списка параметров (рис. 3.1).

**Как сконструирован суперкласс**

При использовании наследования вы должны знать, как сконструирован соответ­ствующий родительский класс. Помните, что, когда оно задействуется, от роди­тельского класса наследуется все. Таким образом, вам потребуется очень хорошо знать все данные и поведения родительского класса. Наследование атрибутов до­вольно очевидно. Однако то, как наследуются конструкторы, не так очевидно. После обнаружения ключевого слова new и выделения памяти для объекта пред­принимаются следующие шаги (рис. 3.4).

1. Внутри конструктора происходит вызов конструктора суперкласса соответ­ствующего класса. Если явного вызова конструктора суперкласса нет, то авто­матически вызывается конструктор по умолчанию. При этом вы сможете увидеть соответствующий код, взглянув на байт-коды.

2. Инициализируется каждый атрибут класса объекта. Эти атрибуты являются частью определения класса (переменные экземпляра), а не атрибутами внутри конструктора или любого другого метода (локальные переменные). В коде DataBaseReader, показанном ранее, целочисленная переменная startPosition яв­ляется переменной экземпляра класса.

3. Выполняется остальная часть кода внутри конструктора.

**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.

Виртуальные и статические методы

Место для ввода текста.

Свойства

Место для ввода текста.

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.