Томский государственный университет
Факультет информатики

А.В. Скворцов, Т.Н. Поддубная

Лабораторные работы по курсу «Введение в объектно-ориентированное программирование в Delphi»

Методические указания

Томск – 2005

Методические указания рассмотрены и одобрены методической комиссией факультета информатики.

Декан факультета информатики С.П. Сущенко

Председатель методической комиссии Б.А. Гладких

В методических указаниях представлены лабораторные задания по курсу «Введение в объектно-ориентированное программирование в Delphi», которые выполняются студентами факультета информатики ТГУ в первом семестре 2-го учебного года.

Методические указания содержат задания к трем лабораторным работам и включают в себя теоретический материал, полное описание классов и частичную реализацию методов.

Почтовый адрес: 634050, г. Томск, пр. Ленина, 36, ТГУ, факультет информатики
Телефон: (382-2) 52-94-96
Информационный интернет-сервер: http://www.inf.tsu.ru
Электронная почта: skv@csd.tsu.ru
poddubnaya@inf.tsu.ru

© Скворцов А. В., Поддубная Т. Н. 2005

МЕТОДИЧЕСКИЕ УКАЗАНИЯ к лабораторной работе №1
по курсу «Объектно-ориентированное программирование»

ЗАДАНИЕ: создать класс **TVector** и написать для него методы (процедуры и функции), реализующие стандартные операции для векторов.

В Delphi есть зарезервированное слово **class**, которое позволяет описывать класс. При создании в Delphi нового проекта в модуле **unit1** появляется объявление **класса основной формы**. Ниже приведен пример стандартного объявления класса:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

По негласному соглашению программисты Delphi начинают имя типа класса с большой буквы T, а имя поля (что будет использовано далее) – с большой буквы F. После служебного слова **class** в круглых скобках указывается базовый класс, от которого порождается данный.

Раздел **public** (открытый) предназначен для объявлений, которые доступны внешнему миру, а в разделе **private** можно объявлять переменные, процедуры и функции, используемые только внутри данного класса.

Подобным образом можно описать класс TVector, необходимый для выполнения первой лабораторной работы. Для примера приведем следующий фрагмент объявления.

```
type
   TVector = class
   private
     FValues: array of double;
//Далее идут 4 accessing метода, которые, в соответствии
// с принципом инкапсуляции, должны быть объявлены в разделе private
     function GetLength: integer;
     procedure SetLength(const Value: integer);
     function GetValues(Index: integer):double;
     procedure SetValues(Index: integer; const Value: double);

  public
    //Receive the vector as string
    function AsString: string;
```

```
//Length of the vector
property Length: integer read GetLength write SetLength;

//Receive the product of the vector to scalar
procedure Product( Factor: double); overload;

//Receive the product of two vectors
procedure Product( const SecondVector: TVector); overload;

end;
```

Разумное использование комментариев, а также мнемонических имен делает текст достаточно понятным.

В приведенном объявлении указаны некоторые составляющие объявления класса: **поля и методы.** Поле FValues объявлено как **динамический** массив вещественных чисел, т.е. его длину можно произвольно менять по ходу исполнения программы.

В разделе private объявлены методы доступа, начинающиеся со слов **Get** и **Set**. Способ написания имен тоже принят программистским сообществом.

В разделе public объявлены имена некоторых необходимых методов в виде процедур и функций, а также **свойство** (служебное слово **property**) класса **Length** (число элементов в массиве) и действия, связанные с чтением и записью значения этого атрибута. Можно было бы назвать это свойство другим именем (например, size), но мы намеренно взяли имя Length, для того чтобы показать, как выходить из положения, когда используемое имя совпадает с системным (см. далее).

Обращаем также внимание на то, что в разделе public объявлены два метода с одинаковым именем **Product,** и каждое из них заканчивается служебным словом **overload** (перегрузка). Эта новая возможность позаимствована, начиная с Delphi 4, из C++. Она разрешает иметь несколько функций и процедур с одинаковым именем. Однако у них должны быть разные наборы параметров или их типы. Тип возвращаемого результата не может служить различием. В нашем случае перегружается метод **Product**. В одном случае вычисляется произведение вектора на некоторое значение, а в другом – поэлементное произведение двух векторов.

Приведенное объявление помещается в разделе **interface** модуля Delphi, а текст объявленных методов (процедур и функций) – в разделе **implementation.** Для рассматриваемого примера код соответствующего модуля будет выглядеть так:

```
unit Vectors;

interface

uses
  SysUtils, Classes;

Type
  //Vector
  TVector = class

  private

     FValues: array of double;
     function GetLength: integer;
     procedure SetLength(const Value: integer);
     function GetValues(Index: integer):double;
     procedure SetValues(Index: integer; const Value: double);
     procedure CheckIndex(Index: integer);

  public

      constructor Create;

     //Get vector values as string
     function AsString: string;

     //Item values of the vector
     property Values[Index: integer]: double
                           read GetValues Write SetValues; default;

     //Length of the vector
     property Length: integer read GetLength write SetLength;

    //Receive the product of the vector to scalar
    procedure Product( Factor: double); overload;

    //Receive the product of two vectors
    procedure Product( const SecondVector: TVector); overload;

end;
```

```
implementation

{TVector}

constructor TVector.Create;
begin
  System.SetLength(Fvalues, 1);
end;

function TVector.AsString: string;
var i: integer;
begin
  Result := '(';
  for i:=0 to Length-1 do
  begin
    if i>0 then Result:= Result+' , ';
    Result:=Result+Format('%.3n',[Values[i]]);
  end;
  Result:=Result+')';
end;

function TVector.GetLength: integer;
begin
  Result:=System.Length(FValues);
end;

function TVector.GetValues(Index: integer): double;
begin
  CheckIndex(Index);
  Result:=FValues[Index];
end;

procedure TVector.SetLength(const Value: integer);
begin
  if Value<1 then
   raise Exception.Create('Invalid vector length');
  System.SetLength(FValues, Value);
end;

procedure TVector.SetValues(Index: integer; const Value: double);
begin
  CheckIndex(Index);
  FValues[Index]:=Value;
end;
```

```
procedure TVector.CheckIndex(Index: integer);
begin
  if (Index>=0) and (Index>=Length) then
    raise Exception.Create('Index out of the bounds');
end.

Procedure TVector.Product(const SecondVector: TVector);
var i: integer;
Begin
   For i:=0 to SecondVector.Length-1 do
     Fvalues[I]:= Self[i] * SecondVector[i];
 end;

end.
```

Задания к работе:

1. Изучить назначение метода **construtor.**
2. Написать методы:
   - для сложения векторов (TVector.Add(const SecondVector: TVector)),
   - умножения вектора на значение (TVector.Product(Factor: double)),
   - получения скалярного произведения векторов.
3. Создать форму для отображения результатов работы проекта.

Ниже приведен возможный вариант реализации п.3

```
unit UnitMain;

interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    ButtonInput: TButton;
    Memo1: TMemo;
    ButtonExit: TButton;
    procedure ButtonInputClick(Sender: TObject);
    procedure ButtonExitClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

uses Vectors;

{$R *.dfm}

procedure TForm1.ButtonInputClick(Sender: TObject);
var  V, SecondV: TVector;
       i, N: integer; R: double;
begin
  //Создание экземпляров класса
  V:=TVector.Create;  SecondV:=TVector.Create;
  try
    V.Length:=4; SecondV.Length:= 4;
    for i:=0 to V.Length-1 do begin
      Randomize;
      V[i]:=Random(100);
    end;
    for i:=0 to SecondV.Length-1 do
      SecondV[i]:= 10;
    Memo1.Lines.add(V.AsString);
    V.Product(10);
    Memo1.Lines.add(V.AsString);
     V.Add(SecondV);
     Memo1.Lines.Add(V.AsString);
     R:=V.ScalarProduct(SecondV);
     Memo1.Lines.Add(FloatToStr(R));
  finally
    V.Free;   SecondV.Free;
  end;
end;

procedure TForm1.ButtonExitClick(Sender: TObject);
begin
  Close;
end;
end.
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ к лабораторной работе № 2
по курсу «Объектно-ориентированное программирование»

ЗАДАНИЕ: создать класс **TMatrix,** в котором можно было бы рассматривать матрицу, как массив векторов (столбцов или строк) и написать для него методы (процедуры и функции), реализующие стандартные операции для матриц.

Для выполнения задания необходимо познакомиться с новыми понятиями объектно-ориентированного программирования в среде Delphi. К ним относятся: **свойства** (***properties***), **виртуальные** (***virtual***) и **абстрактные** (***abstract***) методы.

Кроме того, потребуется изучить особенности реализации в Delphi основных идей объектно-ориентированного программирования – **инкапсуляции** (***incapsulation***), **полиморфизма** (***polymorphism***), **перекрытия** (***overriding***) методов.

ОСНОВНАЯ ЦЕЛЬ ДАННОЙ РАБОТЫ ЗАКЛЮЧАЕТСЯ В ДЕМОНСТРАЦИИ СУЩНОСТИ ПОЛИМОРФИЗМА.

Работа будет строиться как развитие предыдущей работы, в которой создавался класс **TVector.** Дело в том, что строки и столбцы объектов типа **TMatrix** могут вести себя как вектора типа **TVector,** причем в оперативной памяти для **TMatrix** при этом не нужно создавать экземпляры класса **TVector**. То есть, можно сказать, что матрица при необходимости может просто «прикидываться» массивом векторов (строк или столбцов).

Для удобства объяснения на рис. 1 приведена диаграмма, представляющая собой схему иерархии создаваемых классов. Если исходить из предыдущей работы, в которой создавался класс **TVector,** можно сказать, что этот класс разделен на две части – на два класса.

В базовый класс, названный **TCustomVector,** помещены описания (прототипы) методов и свойства, которые позволяют называть его вектором. Это **свойства** – количество элементов в векторе (property Length) и собственно массив значений вектора (property Values). Сюда же можно поместить методы для выполнения различных математических операций над векторами (сложения, умножения и т.д.). Но сам класс создан таким образом, что внутри него НЕ БУДУТ храниться никакие значения. Методы доступа (получения и установки) к элементам массива определены в классе «виртуальными и абстрактными», т.е. недоопределенными. Такие виртуальные методы нужно будет определить в потомках этого класса, которые могут быть как обычными векторами, так и столбцами (строками) в матрице. По сути, объявление класса **TCustomVector** является некоторым «интерфейсом», через который можно, при желании, обращаться к любому вектору, вне зависимости от его физической реализации.

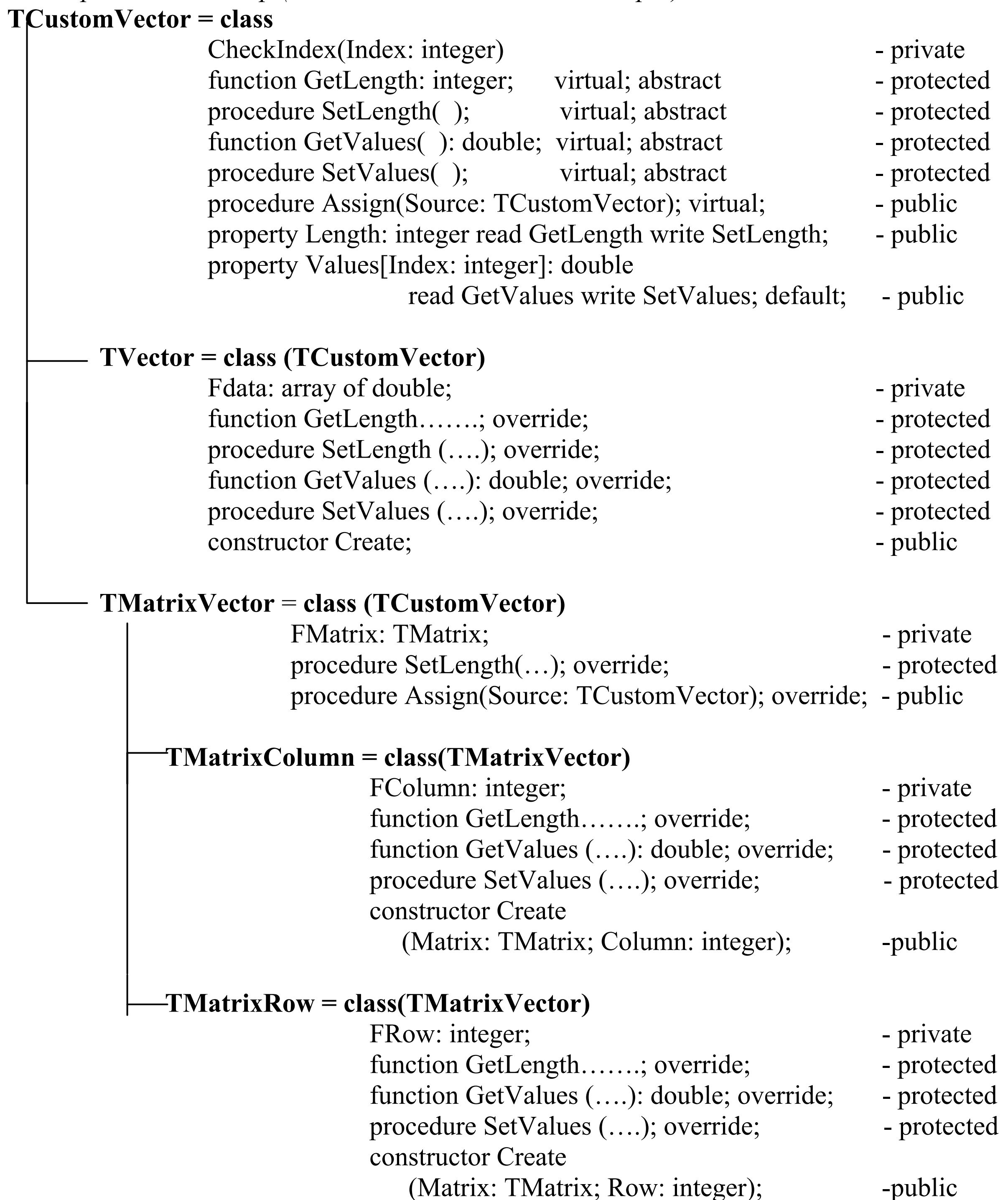

рис. 1

С правой стороны диаграммы указаны категории, к которым отнесены соответствующие компоненты в объявлениях классов.

Как следует из диаграммы, от класса **TCustomVector** унаследованы подклассы **TVector** (обычный вектор, который создавался в предыдущей работе) и подклассы для столбцов и строк матрицы. В подклассе **TVector** введена внутренняя (private) переменная FData в виде динамического массива вещественных значений для хранения значений элементов вектора и доопределены методы доступа к значениям TVector так, чтобы при

обращении к элементам вектора на самом деле происходило обращение к значениям внутреннего динамического массива FData.

От класса **TCustomVector** также унаследованы специальные классы **TMatrixColumn** и **TMatrixRow**, предназначенные для того, чтобы создавать программисту видимость, будто строки и столбцы матрицы являются векторами **TCustomVector**. Для этого необходимо переопределить методы доступа к элементам вектора в этих классах так, чтобы на самом деле обращение происходило к элементам матрицы. Определение этих классов тоже разбито на две части. Для удобства программирования общие методы (SetLength и Assign) и поле FMatrix объявлены и доопределены в классе **TMatrixVector.**

Теперь обратимся к определению класса **TMatrix.**

```
//Матрица
  TMatrix = class
  private
    FColumnCount: integer;
    FRowCount: integer;
    FValues: array of array of double;
    FColumns: array of TCustomVector;
    FRows: array of TCustomVector;
    function GetValues(Column, Row: integer): double;
    procedure SetValues(Column, Row: integer; Value: double);
    procedure SetColumnCount(const Value: integer);
    procedure SetRowCount(const Value: integer);
    procedure CheckIndex(Column,Row: integer);
    procedure SetSizes(NewColumnCount, NewRowCount: integer);
    function GetColumns(Column: integer): TCustomVector;
    function GetRows(Row: integer): TCustomVector;
    procedure SetColumns(Column: integer; const Value: TCustomVector);
    procedure SetRows(Row: integer; const Value: TCustomVector);

  public
    constructor Create;
    //Скопировать размеры матрицы и значения элементов из другой матрицы
    procedure Assign(Source: TMatrix);
    //Получить описание матрицы в виде текста
    function AsString: string;
    //Количество столбцов матрицы
    property ColumnCount: integer read FColumnCount write SetColumnCount;
    //Количество строк матрицы
    property RowCount: integer read FRowCount write SetRowCount;
    //Элементы матрицы (нумерация идет от 0 до ColumnCount-1, RowCount-1);
    property Values[Column, Row: integer]: double  read GetValues write SetValues; default;
    //Столбцы матрицы в виде векторов
    property Columns[Column: integer]: TCustomVector read GetColumns write SetColumns;
    //Строки матрицы в виде векторов
    property Rows[Row: integer]: TCustomVector read GetRows write SetRows;
  end;
```

Класс **TMatrix** имеет простую структуру, без виртуальных методов. Он включает в себя свойства ширины и высоты матрицы (количество столбцов и строк), индексированное свойство для доступа к элементам матрицы (Values), свойства, представляющие столбцы и строки матрицы в виде векторов. В разделе private объявлены соответствующие методы доступа, а в разделе public – методы, использующиеся при написании алгоритмов с матрицами. В приведенном выше объявлении класса комментарии помогут понять смысл объявления каждого атрибута класса.

Несколько слов о появившихся в Delphi атрибутах класса – **свойствах** (**properties**). Они определяют состояние и поведение объекта. Свойство – это просто имя, связанное с методами чтения и записи данных или обращающееся к данным напрямую. Другими словами, при чтении или изменении свойства происходит обращение к полю или вызывается соответствующий метод. Благодаря технологии Code Completion, применяемой в Delphi, написание методов, указанных в свойствах, облегчается, т.к. Delphi сама может создавать для них необходимые шаблоны. После того, как имя свойства набрано в объявлении класса, достаточно нажать комбинацию клавиш **Ctrl+Shift+C**, и нужная заготовка появится, потребуется лишь только добавить в нее соответствующий код. Так как свойства скрывают от пользователя внутреннюю реализацию доступа к информации (либо прямой доступ, либо с помощью вызова метода), их можно считать приложением концепции **инкапсуляции** объектно-ориентированного программирования.

Класс **TCustomVector** в данной задаче используется только для порождения подклассов. Такой класс называется **абстрактным;** он не имеет экземпляров и содержит объявления методов с ключевым словом **abstract**, функциональность которых определяется только в классах-потомках.

Для изменения (**перекрытия**) какого-либо метода родительского класса в классе-потомке он сопровождается ключевым словом **virtual** в родительском классе, а в подклассе – ключевым словом **override**.

Суть полиморфизма, изучению которого посвящена данная работа, заключается в том, что все классы TVector, TMatrixColumn и TMatrixRow унаследованы от одного базового класса TCustomVector, и поэтому с ними можно работать унифицированно, как с обычными векторами. Например, можно выполнить одной командой сложение строки и столбца в матрице TMatrix, или умножить строку матрицы TMatrix на любой другой вектор типа TVector.

Далее приведен неполный текст модуля для выполнения поставленной задачи. Добавьте в указанные в нем процедуры и функции недостающий код.

Напишите методы, реализующие математические операции с матрицами, используя разные способы обращения к элементам матрицы.

```
unit Matrices;

interface

Uses
  SysUtils, Classes;

type
  //Абстрактный вектор (нельзя создавать его экземпляры)
  TCustomVector = class
  private
    procedure CheckIndex(Index: integer);
  protected
    function GetLength: integer; virtual; abstract;
    procedure SetLength(NewLength: integer); virtual; abstract;
    function GetValues(Index: integer): double; virtual; abstract;
    procedure SetValues(Index: integer; NewValue: double); virtual; abstract;
  public
    //Скопировать размер вектора и его значения из другого вектора
    procedure Assign(Source: TCustomVector); virtual;
    //Длина вектора
    property Length: integer read GetLength write SetLength;
    //Элементы вектора (нумерация идет от 0 до Length-1)
    property Values[Index: integer]: double
                          read GetValues write SetValues; default;
  end;


  //Обычный вектор
  TVector = class(TCustomVector)
  private
    FData: array of Double;
  protected
    function GetLength: integer; override;
    procedure SetLength(NewLength: integer); override;
    function GetValues(Index: integer): double; override;
    procedure SetValues(Index: integer; NewValue: double); override;
  public
    constructor Create;
  end;
```

```
  //Матрица
  TMatrix = class
  private
    FColumnCount: integer;
    FRowCount: integer;
    FValues: array of array of double;
    FColumns: array of TCustomVector;
    FRows: array of TCustomVector;
    function GetValues(Column, Row: integer): double;
    procedure SetValues(Column, Row: integer; Value: double);
    procedure SetColumnCount(const Value: integer);
    procedure SetRowCount(const Value: integer);
    procedure CheckIndex(Column,Row: integer);
    procedure SetSizes(NewColumnCount, NewRowCount: integer);
    function GetColumns(Column: integer): TCustomVector;
    function GetRows(Row: integer): TCustomVector;
    procedure SetColumns(Column: integer; const Value: TCustomVector);
    procedure SetRows(Row: integer; const Value: TCustomVector);
  public
     constructor Create;
     //Скопировать размеры матрицы и значения элементов из другой матрицы
     procedure Assign(Source: TMatrix);
     //Получить описание матрицы в виде текста
     function AsString: string;
     //Количество столбцов матрицы
     property ColumnCount: integer read FColumnCount write SetColumnCount;
     //Количество строк матрицы
     property RowCount: integer read FRowCount write SetRowCount;
     //Элементы матрицы (нумерация идет от 0 до ColumnCount-1, RowCount-1);
     property Values[Column, Row: integer]: double
                        read GetValues write SetValues; default;
     //Столбцы матрицы в виде векторов
     property Columns[Column: integer]: TCustomVector
                        read GetColumns write SetColumns;
     //Строки матрицы в виде векторов
     property Rows[Row: integer]: TCustomVector read GetRows write SetRows;
  end;


implementation

{ TCustomVector }
procedure TCustomVector.Assign(Source: TCustomVector);
var i: integer;
```

```
begin
  if Source<>nil then
  begin
    Length:=Source.Length;
    for i:=0 to Length-1 do
      Values[i]:=Source[i];
  end;
end;

procedure TCustomVector.CheckIndex(Index: integer);
begin
  if(Index<0) or (Index>=Length) then
    raise Exception.Create('Недопустимый индекс элемента вектора');
end;

{ TVector }
constructor TVector.Create;
begin
  inherited Create;
  System.SetLength(FData, 1);
end;

function TVector.GetLength: integer;
begin
  //Добавить код
end;

function TVector.GetValues(Index: integer): double;
begin
     //Добавить код
end;

procedure TVector.SetLength(NewLength: integer);
begin
    //Добавить код
end;



procedure TVector.SetValues(Index: integer; NewValue: double);
begin
   //Добавить код
end;
```

//Далее следует инкапсулированное в класс TMatrix объявление типов.

```
type

  //Строка или столбец матрицы
  TMatrixVector = class(TCustomVector)
  private
    FMatrix: TMatrix;
  protected
    procedure SetLength(NewLength: integer); override;
  public
    //Скопировать размер вектора и его значения из другого вектора
    procedure Assign(Source: TCustomVector); override;
  end;

  //Столбец матрицы
  TMatrixColumn = class (TMatrixVector)
  private
    FColumn: integer;
  protected
    function GetLength: integer; override;
    function GetValues(Index: integer): double; override;
    procedure SetValues(Index: integer; NewValue: double); override;
  public
    constructor Create(Matrix: TMatrix; Column: integer);
  end;

  //Строка матрицы
  TMatrixRow = class (TMatrixVector)
  private
    FRow: integer;
  protected
    function GetLength: integer; override;
    function GetValues(Index: integer): double; override;
    procedure SetValues(Index: integer; NewValue: double); override;
  public
    constructor Create(Matrix: TMatrix; Row: integer);
  end;


  { TMatrixVector }
  //Скопировать размер вектора и его значения из другого вектора
  procedure TMatrixVector.Assign(Source: TCustomVector);
  var i: integer;
  begin
```

```
  if Source.Length<>Length then
    raise Exception.Create('Недопустимо выполнять копирование значений для векторов разной длины. ');
  for i:=0 to Length-1 do
    Values[i]:=Source[i];
end;

procedure TMatrixVector.SetLength(NewLength: integer);
begin
  if NewLength<>Length then
    raise Exception.Create('Изменение размеров вектора, являющегося элементом матрицы, запрещено.');
end;

{ TMatixColumn }

constructor TMatrixColumn.Create(Matrix: TMatrix; Column: integer);
begin
  inherited Create;
  FMatrix:=Matrix;
  FColumn:=Column;
end;

function TMatrixColumn.GetLength: integer;
begin
  Result:=FMatrix.RowCount;
end;

function TMatrixColumn.GetValues(Index: integer): double;
begin
  Result:=FMatrix[FColumn, Index];
end;

procedure TMatrixColumn.SetValues(Index: integer; NewValue: double);
begin
  FMatrix[FColumn, Index]:=NewValue;
end;



 { TMatrixRow }
constructor TMatrixRow.Create(Matrix: TMatrix; Row: integer);
  begin
   //Добавить код
  end;
```

```
function TMatrixRow.GetLength: integer;
  begin
     //Добавить код
  end;

function TMatrixRow.GetValues(Index: integer): double;
  begin
     //Добавить код
  end;

procedure TMatrixRow.SetValues(Index: integer; NewValue: double);
  begin
    //Добавить код
  end;


 { TMatrix }
constructor TMatrix.Create;
 begin
   inherited Create;
   SetSizes(1,1);
 end;

procedure TMatrix.CheckIndex(Column, Row: integer);
 begin
   if (Column<0)or(Column>=ColumnCount) then
     raise Exception.Create('Недопустимый номер столбца матрицы');
   if (Row<0)or(Row>=RowCount) then
     raise Exception.Create('Недопустимый номер строки матрицы');
 end;

procedure TMatrix.SetValues(Column, Row: integer; Value: double);
begin
  CheckIndex(Column, Row);
  FValues[Column, Row]:=Value;
end;

function TMatrix.GetValues(Column, Row: integer): double;
begin
   CheckIndex(Column, Row);
   Result:=FValues[Column, Row];
 end;
```

```
function TMatrix.AsString: string;
var c, r: integer;
begin
//Добавить код
 end;

procedure TMatrix.SetColumnCount(const Value: integer);
begin
  SetSizes(Value, RowCount);
end;

procedure TMatrix.SetRowCount(const Value: integer);
begin
  SetSizes(ColumnCount, Value);
end;

procedure TMatrix.SetSizes(NewColumnCount, NewRowCount: integer);
var i: integer;
begin
  if NewColumnCount<1 then
    raise Exception.Create('Число столбцов матрицы должно быть положительным.');
  if NewRowCount<1 then
    raise Exception.Create('Число строк матрицы должно быть положительным.');
  for i:=NewColumnCount to ColumnCount-1 do
    FreeAndNil(FColumns[i]);
  for i:=NewRowCount to RowCount-1 do
    FreeAndNil(FRows[i]);
  FColumnCount:=NewColumnCount;
  FRowCount:=NewRowCount;
  System.SetLength(FValues, ColumnCount, RowCount);
  System.SetLength(FColumns, ColumnCount);
  System.SetLength(FRows, RowCount);
end;

function TMatrix.GetColumns(Column: integer): TCustomVector;
begin
  CheckIndex(Column, 0);
  if FColumns[Column]=nil then
    FColumns[Column]:=TMatrixColumn.Create(Self, Column);
  Result:= FColumns[Column];
end;
```

```
  function TMatrix.GetRows(Row: integer): TCustomVector;
  begin
    CheckIndex(0, Row);
    if FRows[Row]=nil then
      FRows[Row]:=TMatrixRow.Create(Self, Row);
    Result:= FRows[Row];
  end;

  procedure TMatrix.SetColumns(Column: integer; const Value: TCustomVector);
  begin
    Columns[Column].Assign(Value);
  end;

  procedure TMatrix.SetRows(Row: integer; const Value: TCustomVector);
  begin
    Rows[Row].Assign(Value);
  end;

  procedure TMatrix.Assign(Source: TMatrix);
  var i,j: integer;
  begin
    if Source=nil then exit;
    SetSizes(Source.ColumnCount, Source.RowCount);
    for i:=0 to ColumnCount-1 do
      for j:=0 to RowCount-1 do
        Values[i,j]:=Source[i,j];
   end;

end.
```

Ниже в модуле «unit Main» приведен вариант выполнения задания.

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    ButtonCreate: TButton;
```

```
    ButtonExit: TButton;
    procedure ButtonExitClick(Sender: TObject);
    procedure ButtonCreateClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

uses UnitMatr;

{$R *.dfm}

procedure TForm1.ButtonExitClick(Sender: TObject);
begin
  Close;
end;

procedure TForm1.ButtonCreateClick(Sender: TObject);
var
  M: TMatrix;
  i,j: integer;
begin
  Randomize;
  M:=TMatrix.Create;
  try
      //Задаем размеры матрицы M
    M.ColumnCount:=3; M.RowCount:=2;
      //Заполняем матрицу M случайными числами
     // Добавить код
      //Выводим в поле Memo1 матрицу M и элемент M[0,1]
    Memo1.Lines.Clear;
    Memo1.Lines.Add('Матрица M:');
    Memo1.Lines.Add(M.AsString);
    Memo1.Lines.Add('элемент на пересеч. 0 столбца и 1 строки матрицы M:');
    Memo1.Lines.Add(FloatToStr(M.Columns[1][0]));
  finally
    M.Free;
  end;
end;  end.
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ к лабораторной работе № 3
по курсу «Объектно-ориентированное программирование»

ЗАДАНИЕ: в качестве примера объектно-ориентированного программирования создать астрономическую модель, представляющую собой Солнечную систему. Модель должна работать следующим образом: на экране открывается окно, и в нем появляется Солнце и планеты со своими спутниками на соответствующих астрономических местах. Планеты начинают вращаться вокруг Солнца по своим орбитам с правильным соотношением скоростей. В это же время спутники начинают вращаться вокруг своих планет по траекториям, складывающимся из двух вращательных движений: вращения планеты вокруг Солнца и вращения спутника вокруг своей планеты.

В качестве варианта можно воспользоваться следующей схемой иерархии классов:

```
//Абстрактный класс (нельзя создавать его экземпляры)
TBody = class
 |        procedure StandardDraw
 |                 (Canvas: TCanvas; Color:Tcolor; Radius: integer )         - public
 |        procedure GetXY(Out X,Y: double); Virtual;  Abstract;              - public
 |        procedure Draw(Canvas: TCanvas);   Virtual;  Abstract;             - public
 |        procedure Move;                     Virtual;  Abstract;            - public
 |
 |------ TStar = class (TBody)
 |              Fx: double;                                                   - private
 |              Fy: double;                                                   - private
 |              procedure GetXY(Out GX, GY: double); override;                - public
 |              procedure Draw(Canvas: TCanvas);   override;                  - public
 |              procedure Move;                     override                  - public
 |              property X: double read Fx;                                   - public
 |              property Y: double read Fy;                                   - public
 |
 |------ TSatellite = class (TBody)
 |          |     ParentBody: TBody;                                          - private
            |     R: double;                                                  - private
            |     Alpha: double;                                              - private
            |     DAlpha: double;                                             - private
            |     procedure GetXY(Out X, Y: double); override;                - public
            |     procedure Move;                     override;               - public
            |
            |---TPlanet= class(TSatellite)
            |         procedure Draw(Canvas: TCanvas);   override;            - public
            |
            |
            |---TSputnic = class(TSatellite)
            |          procedure Draw(Canvas: TCanvas);   override;           - public
                       Constructor Create(GR: double);                        - public
```

```
unit UnitPlanets;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Math, ExtCtrls, StdCtrls;
        Type
  //Абстрактный класс
  TBody = class
  public
    procedure StandardDraw(Canvas: TCanvas; Color: TColor; Radius: integer);
    procedure GetXY(Out X,Y: double); Virtual; Abstract;
    procedure Draw(Canvas: TCanvas); Virtual; Abstract;
    procedure Move(Canvas: TCanvas); Virtual; Abstract;
  end;
  TStar = class (TBody)
  private
    //ParentBody: TBody;
    Fx: double;
    Fy: double;
  public
    procedure GetXY(Out GX, GY: double); override;
    procedure Draw(Canvas: TCanvas); override;
    procedure Move(Canvas: TCanvas); override;
    property X: double read Fx;
    property Y: double read Fy;
  end;

  TSatellite = class(TBody)
  private
    ParentBody: TBody;
    R: double;
    Alpha: double;   //Угловое перемещение тела
    DAlpha: double;
  public
    procedure GetXY(Out X,Y: double); override;
    procedure Move(Canvas: TCanvas); override;
  end;

  TPlanet = class(TSatellite)
    public
    procedure Draw(Canvas: TCanvas); override;
  end;
```

```
  TSputnic = class(TSatellite)
  public
    procedure Draw(Canvas: TCanvas); override;
    Constructor Create(GR: double);
  end;

type
  TSkyForm = class(TForm)
    Image1: TImage;
    ButtonExit: TButton;
    Timer1: TTimer;
    procedure ButtonExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormResize(Sender: TObject);

  private
    { Private declarations }
    Bodies: TList;    //Список для хранения созданных объектов
  public
    { Public declarations }
  end;

var
  SkyForm: TSkyForm;

implementation

{$R *.dfm}

{ TStar }

procedure TStar.Draw(Canvas: TCanvas);
begin
  StandardDraw(Canvas, clYellow, 20);
end;

procedure TStar.GetXY(out GX, GY: double);
begin
  GX:=X; GY:=Y;
end;

procedure TStar.Move(Canvas: TCanvas);
begin
  Fx:=Fx+0; Fy:=Fy+0;          //Солнце не движется
```

```
  {Fx:=Fx+Fdx; Fy:=Fy+Fdy;}
end;

{ TSatellite }

procedure TSatellite.GetXY(out X, Y: double);
var si,co: extended;
begin
  ParentBody.GetXY(X,Y);
  SinCos(Alpha, si, co);
  X:=X+R*co; Y:=Y+R*si;
end;

procedure TSatellite.Move(Canvas: TCanvas);
//var X,Y: integer; XX, YY: double;
begin
  Alpha:=Alpha+DAlpha;
end;

{ TPlanet }

procedure TPlanet.Draw(Canvas: TCanvas);
begin
  StandardDraw(Canvas, clGreen, 10);
end;

{ TSputnic }
constructor TSputnic.Create(GR: double);
begin
   inherited Create;
   R:=GR;
end;

procedure TSputnic.Draw(Canvas: TCanvas);
begin
  StandardDraw(Canvas, clBlue, 5);
end;
{ TBody }
procedure TBody.StandardDraw(Canvas: TCanvas; Color: TColor;
  Radius: integer);
var X,Y: integer; XX,YY: double;
begin
  with Canvas do
  begin
    Pen.Color:=clBlack; Pen.Width:=1;
```

```
    Brush.Style:=bsSolid; Brush.Color:=Color;
    GetXY(XX,YY);
    X:=Round(XX); Y:=Round(YY);
    ellipse (X-Radius, Y-Radius, X+Radius, Y+Radius);
  end;
end;

procedure TSkyForm.ButtonExitClick(Sender: TObject);
begin
  Close;
end;

procedure TSkyForm.FormCreate(Sender: TObject);
var Sun: TStar; TheEarth, Mars: TPlanet;  Moon: TSputnic;
begin
  Bodies:=TList.Create;
  Sun:=TStar.Create;
  Sun.Fx:=250; Sun.Fy:=200;
  Bodies.Add(Sun);

  TheEarth:=TPlanet.Create;
  TheEarth.ParentBody:=Sun;
  TheEarth.R:=90; TheEarth.DAlpha:=0.01;
  Bodies.Add(TheEarth);

  Moon:= TSputnic.Create(25);
  Moon.ParentBody:=TheEarth;
  Moon.DAlpha:=0.1;
  Bodies.Add(Moon);

  Mars:=TPlanet.Create;
  Mars.ParentBody:=Sun;
  Mars.R:=150; Mars.DAlpha:=0.03;
  Bodies.Add(Mars);
end;

procedure TSkyForm.Timer1Timer(Sender: TObject);
var B: TBody; i: integer;
 begin
   Timer1.Interval:=100;
   with Image1.Picture.Bitmap, Canvas do
   begin
     Brush.Color:=clWhite;
     FillRect(Rect(0,0,Width,Height));
   end;
```

```
  for i:=0 to Bodies.Count-1 do
  begin
    B:=Bodies[i];
    B.Move(Image1.Canvas);
    B.Draw(Image1.Picture.Bitmap.Canvas);
  end;
 Image1.Repaint; end;

procedure TSkyForm.FormResize(Sender: TObject);
begin
  Image1.Picture.Bitmap.Width:=Image1.Width;
  Image1.Picture.Bitmap.Height:=Image1.Height;
end;

end.
```

1. В приведенном выше тексте программы объяснить выделенные жирным шрифтом понятия и процедуры.

2. Добавить в приведенный текст программы возможность для рисования траекторий движущихся объектов.

//Задания подготовлены преподавателями факультета информатики профессором Скворцовым А.В. и доцентом Поддубной Т.Н. в 2004 г.