**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ**

**ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**«ИНСТИТУТ СОЦИАЛЬНЫХ ТЕХНОЛОГИЙ»**



**КУРСОВАЯ РАБОТА**

**На тему: «Объектно-ориентированное программирование в среде Lazarus»**

Выполнили: студенты группы И-72

Слутских Дмитрий и Белов Александр

Руководитель:

Сафаров Р.А.

Новосибирск

2020

Содержание

[Введение 2](#_Toc39253006)

[Цели 3](#_Toc39253007)

[Что такое ООП 4](#_Toc39253008)

[ООП в среде программирования Lazarus 8](#_Toc39253009)

[Заключение 13](#_Toc39253010)

[Приложение 14](#_Toc39253011)

# Введение

Данная курсовая работа направленна на то, чтобы рассказать, объяснить основные принципы написания программ с помощью ООП (Объектно-ориентированное программирование).

Основополагающая программа курсовой работы была сделана в среде Lazarus на языке Pascal. Так как Lazarus достаточно простая среда программирования на Pascal.

Lazarus — открытая среда разработки программного обеспечения на языке Object Pascal для компилятора Free Pascal (часто используется сокращение FPC — Free Pascal Compiler, бесплатно распространяемый компилятор языка программирования Pascal).

Интегрированная среда разработки предоставляет возможность кроссплатформенной разработки приложений в Delphi-подобном окружении. Что позволяет достаточно несложно переносить Delphi-программы с графическим интерфейсом в различные операционные системы: Linux, FreeBSD, Mac OS X, Microsoft Windows, Android.

Среда разработки Lazarus предоставляет доступ только к тем участкам кода, с которыми необходимо работать, скрывая основную часть программы, которая создается автоматически самой средой разработки.

Pascal в свою очередь является наиболее универсальным и легко изучаемым языком, среди множества других. И Pascal позволяет использовать многие возможности программирования.

# Цели

Основными целями работы являются:

* Рассказать, что такое ООП (Объектно-ориентированное программирование).
* Рассказать и объяснить основные принципы ООП (Объектно-ориентированное программирование) в среде Lazarus.
* Написать и представить программу с использование ООП (Объектно-ориентированное программирование).

# Что такое ООП

**ООП** – это методика разработки программы, в основе которой лежит понятие объекта как некоторой структуры, описывающей объект реального мира, его поведение и взаимодействие с другими объектами.

Объектно-ориентированное программирование (ООП) представляет собой этап развития концепций построения языков программирования. Здесь получили дальнейшее развитие принципы структурного программирования - структуризация программ и данных, модульность и т. д. В основе ООП лежит понятие объекта (object).

**Объект** - это специальный тип записи. Запись объекта содержит все поля, объявленные в определении объекта (как и в обычной записи), но в дополнение могут быть объявлены процедуры и функции, как неотъемлемые части этой записи, указывающие на методы, ассоциированные с типом объекта.

Объект состоит из трех основных частей:

1. Имя (например, автомобиль);
2. Состояние или переменные состояния (например, марка автомобиля, цвет, масса, число мест и т. д.);
3. Методы, или операции, которые выполняют некоторые действия над объектами и определяют, как объект взаимодействует с окружающим миром.

Пример объекта - это массив вещественных значений, совместно с методом для вычисления среднего арифметического.

Type

Average = Object

NumVal: Integer;

Values: Array [1..200] of Real;

Function Mean: Real; { calculates the average value of the array }

End;

Объекты могут “наследовать” поля и методы от “родительских” объектов. Этим подразумевается, что эти поля и методы могут применяться и в объекте, объявленном “потомком”. Но об этом будет сказано позже при описании одного из принципов ООП “наследования”.

Для работы с объектами во FreePascal введено понятие класса. Класс – сложная структура, включающая в себя описание данных, процедуры и функции, которые могут быть выполнены над объектом.

Классом называется составной тип данных, членами (элементами) которого являются функции и переменные (поля). В основу понятия «класс» положен тот факт, что "над объектами можно совершать различные операции".

Для объявления класса используется конструкция:

type

<название класса> = class(<имя класса родителя>)

<методы и переменные класса>

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;

type

TCustomPetal = class

Protected

R, phi, PetalI: double;

X, Y, K, CX, CY: integer;

Scale, RColor: integer;

Public

constructor Create (Xmax, Ymax: integer); virtual;

procedure Draw (Canvas: TCanvas; Erase: boolean = FALSE);

end;

* **private(частные)** - это означает, что элементы, определенные здесь, доступны или видимы другим классам или процедурам/функциям, определенным только внутри данного программного модуля. Это как локальные переменные или локально используемые методы но можно использовать или ссылаться на элементы, объявленные в секциях *protected* или *public*.
* **protected(защищенные)** - это означает, что элементы, определенные здесь, доступны или видимы только классам, произведенным из этого класса.
* **public** - это означает, что элементы, определенные здесь, доступны любому программному модулю, включившему текущий модуль в свою секцию *[Uses](https://wiki.freepascal.org/Glossary" \l "Uses" \o "Glossary)*.
* **published** - подобна секции *public*, но компилятор генерирует типовую информацию, требуемую для автоматического подключения этих классов. Часто список published элементов появляется Инспекторе объектов Lazarus; если список *published* отсутствует, в Инспекторе объектов обычно появляются поля *public*.

Свойства объектов описываются с помощью переменных (полей) классов, объекты называются экземплярами класса, а действия над объектами описываются с помощью подпрограмм, которые называются «методами класса».

Метод подобен обычной процедуре или функции, но может иметь некие дополнительные *директивы*.

type

TCustomPetal = class

Protected

R, phi, PetalI: double;

X, Y, K, CX, CY: integer;

Scale, RColor: integer;

Public

constructor Create (Xmax, Ymax: integer); virtual;

procedure Draw (Canvas: TCanvas; Erase: boolean = FALSE);

end;

TPetal = class (TCustomPetal)

Public

constructor Create (Xmax, Ymax: integer); override;

end;

Некоторые методы, определенные выше, помечены директивой *virtual*(*виртуальный*); другие - *override* (*аннулируемый*).

* **virtual** означает, что тип, или фактический экземпляр метода не известен во время компиляции, но выбирается во время выполнения в зависимости от того, какая подпрограмма вызывает метод. Он может рассматриваться как заполнитель в определении класса.
* **override** означает, что во время выполнения локально данное определение может заменить определение, унаследованное от класса - предка, особенно, если оно было *virtual*. Если необходимо применить метод, определенный в родительском классе, его нужно вызывать специфично, с оговоркой *inherited*.

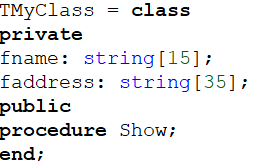
Методы без директив virtual или override являются *статическими* (обычными для Pascal). Методы, имеющие такие директивы, считаются *динамическими*.

# ООП в среде программирования Lazarus

Объектно-ориентированное программирование (ООП) – представляет собой технологию разработки программ с использованием объектов.

В объектно-ориентированных языках есть три основных понятия: *инкапсуляция*, *наследование* и *полиморфизм*.

Важный принцип ООП — ***инкапсуляция***. Это значит, что класс должен скрывать внутри себя всю логику своей работы. Инкапсуляция позволяет создавать класс как нечто целостное, имеющее определённую функциональность. Например, класс ниже имеет поля с разным типом данных, которые определяют уникальные характеристики и методы работы.



Основная идея инкапсуляции – защитить поля от несанкционированного доступа. Поэтому целесообразно поля объявлять в разделе private. Прямой доступ к полям объекта: чтение и обновление их содержимого должно производиться посредством вызова соответствующих методов. В FreePascal для этого служат свойства класса.

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.

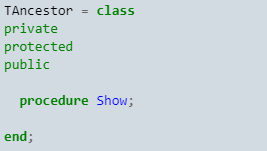


Второй основополагающей составляющей объектно-ориентированного программирования является ***наследование***.

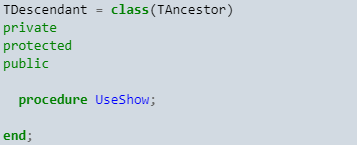
Смысл наследования заключается в следующем: если нужно создать новый класс, лишь немного отличающийся от старого, то нет необходимости в переписывании заново уже существующих полей и методов.

Для этого один из объектов объявляется потомком другого, который, в свою очередь, становится предком этого нового объекта. Потомок наследует все параметры своего предка и его методы, поэтому вторично их описывать нет необходимости, а использовать можно. Это существенно упрощает запись схожих объектов, если установить между ними наследственную связь.

Предок:

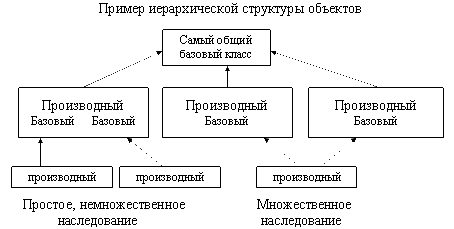


Наследник:



Наследование позволяет создавать иерархию объектов, начиная с некоторого простого первоначального (предка) и заканчивая более сложными, но включающими (наследующими) свойства предшествующих элементов (потомки). Эта иерархия в общем случае может иметь довольно сложную

древовидную структуру.



Каждый потомок несет в себе характеристики своего предка (содержит те же данные и методы), а также обладает собственными характеристиками. При этом наследуемые данные и методы описывать у потомка нет необходимости.

В Object Pascal все классы являются потомками класса TObject. Поэтому если вы создаете дочерний класс прямо от класса TObject, то в определении его можно не писать.

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.

И тут вступает в силу третий основополагающий составляющей объектно-ориентированного программирования ***полиморфизм.***

Если говорить кратко, *полиморфизм* — это способность объекта использовать методы производного класса, который не существует на момент создания базового.

И для использования этого свойства необходимо перегрузить метод 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.

Конструкция полиморфизма, примерно, выглядит так:

(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;*

type

TCustomPetal = class

Protected

R, phi, PetalI: double;

X, Y, K, CX, CY: integer;

Scale, RColor: integer;

Public

constructor Create (Xmax, Ymax: integer); virtual;

procedure Draw (Canvas: TCanvas; Erase: boolean = FALSE);

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;*

type

TPetal = class (TCustomPetal)

Public

constructor Create (Xmax, Ymax: integer); override;

end;

# Заключение

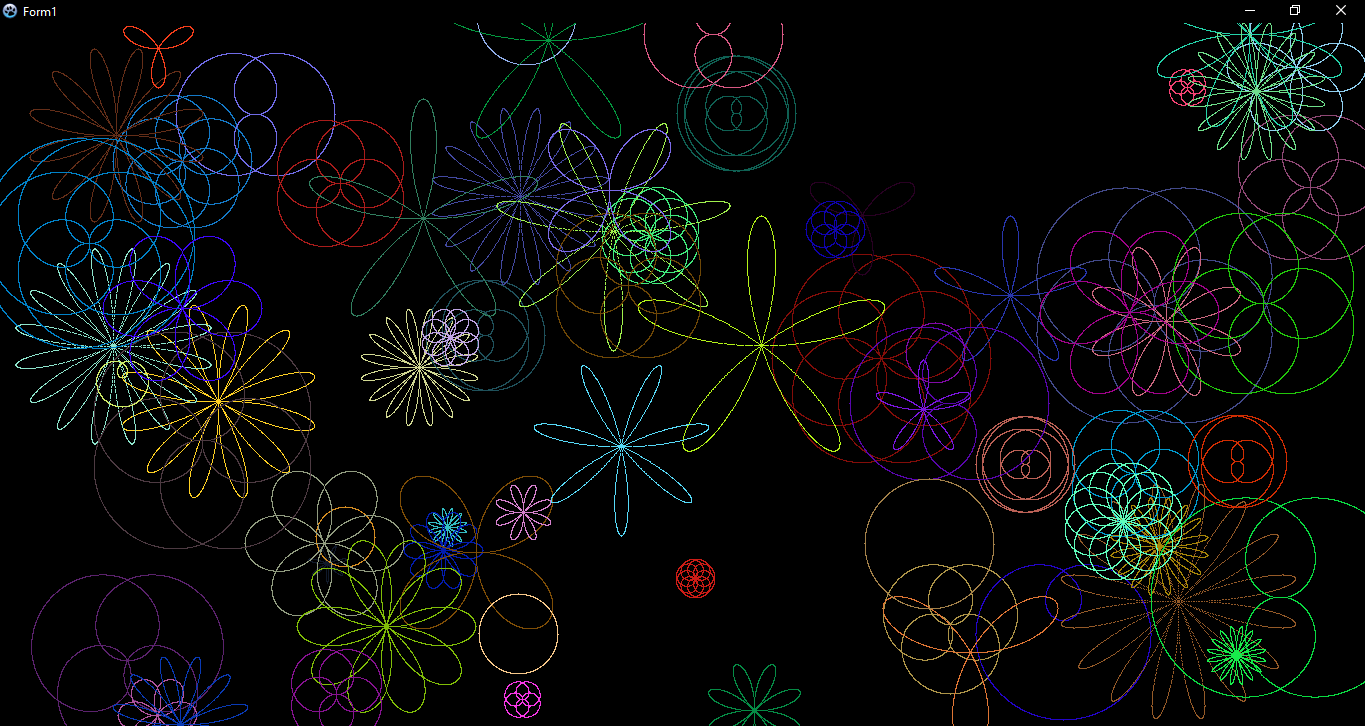
По итогу курсовой работы были выполнены все поставленные цели. Было описано и объяснено, что же такое ООП (Объектно-ориентированное программирование).

Были подробно расписаны основополагающие принципы ООП (Объектно-ориентированное программирование) в среде Lazarus. Такие как, инкапсуляция, наследование и полиморфизм. Так же к подробному описанию и объяснению, для лучшего понимания, были предоставлены примеры этих самых принципов и их составных частей.

А также была написана крупная программа с использование всех принципов ООП (Объектно-ориентированное программирование) в среде Lazarus.

# Приложение

Программа будет рисовать в случайных местах на экране фигуры с разными характеристиками. Потом их же затирать и рисовать новые, и т.д.



unit main;

{$mode objfpc}{$H+}

interface

uses

Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, ExtCtrls,

petalclass;

const

P\_COUNT = 70;

type

{ TForm1 }

TForm1 = class(TForm)

Img: TImage;

Timer1: TTimer;

procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);

procedure FormCreate(Sender: TObject);

procedure FormKeyPress(Sender: TObject; var Key: char);

procedure FormShow(Sender: TObject);

procedure Timer1Timer(Sender: TObject);

private

P: TPetals;

public

end;

var

Form1: TForm1;

implementation

{$R \*.lfm}

{ TForm1 }

procedure TForm1.Timer1Timer(Sender: TObject);

begin

P.DrawNext;

end;

procedure TForm1.FormKeyPress(Sender: TObject; var Key: char);

begin

Timer1.Enabled := False;

Close;

end;

procedure TForm1.FormShow(Sender: TObject);

begin

P := TPetals.Create(img,P\_COUNT);

Timer1.Enabled := True;

end;

procedure TForm1.FormClose(Sender: TObject; var CloseAction: TCloseAction);

begin

P.Clear;

P.Free;

end;

procedure TForm1.FormCreate(Sender: TObject);

begin

end;

end.

===========================================================

unit petalclass;

{$mode objfpc}{$H+}

interface

uses

Classes, SysUtils, Graphics, ExtCtrls, Contnrs;

type

{ TCustomPetal }

TCustomPetal = class

protected

R, phi, PetalI: double;

X, Y, K, CX, CY: integer;

Scale, RColor: integer;

Image: TImage;

public

constructor Create(\_Image: TImage); virtual;

procedure Draw(\_Erase: boolean = False);

procedure Erase;

end;

{ TPetal }

TPetal = class(TCustomPetal)

public

constructor Create(\_Image: TImage); override;

end;

{ TOverlappedPetal }

TOverlappedPetal = class(TCustomPetal)

public

constructor Create(\_Image: TImage); override;

end;

{ TPetals }

TPetals = class(TObjectList)

private

fMaxPetals: smallint;

fImage: TImage;

public

property MaxPetals: smallint read fMaxPetals write fMaxPetals;

constructor Create(\_Image: TImage; \_Size: integer);

procedure DrawNext;

end;

TRandomPetal = class of TCustomPetal;

implementation

{ TPetals }

constructor TPetals.Create(\_Image: TImage; \_Size: integer);

begin

inherited Create;

MaxPetals := \_Size;

fImage := \_Image;

end;

procedure TPetals.DrawNext;

var

LRandomPetal: TRandomPetal;

begin

if self.Count = MaxPetals then

begin

TCustomPetal(First).Erase;

Remove(First);

end;

if Random(2) = 1 then

LRandomPetal := TPetal

else

LRandomPetal := TOverlappedPetal;

Add(LRandomPetal.Create(fImage));

end;

{ TCustomPetal }

constructor TCustomPetal.Create(\_Image: TImage);

begin

inherited Create;

Image:=\_Image;

CX := Random(Image.Canvas.Width);

CY := Random(Image.Canvas.Height);

RColor := 1 + Random($FFFFF0);

Scale := 2 + Random(12);

end;

procedure TCustomPetal.Draw(\_Erase: boolean = False);

var

OldColor: TColor;

begin

phi := 0;

OldColor := RColor;

if \_Erase then

RColor := clBlack;

while phi < K \* pi do

begin

R := 10 \* sin(PetalI \* phi);

X := CX + Trunc(Scale \* R \* cos(phi));

Y := CY - Trunc(Scale \* R \* sin(phi));

if (not \_Erase) or (Image.Canvas.Pixels[X, Y] = OldColor) then

Image.Canvas.Pixels[X, Y] := RColor;

phi += pi / 1800;

end;

end;

procedure TCustomPetal.Erase;

begin

Draw(True);

end;

{ TOverlappedPetal }

constructor TOverlappedPetal.Create(\_Image: TImage);

begin

inherited Create(\_Image);

K := 12;

while PetalI = Round(PetalI) do

PetalI := (1 + Random(6)) / (1 + Random(6));

Draw;

end;

{ TPetal }

constructor TPetal.Create(\_Image: TImage);

begin

inherited Create(\_Image);

K := 2;

PetalI := 1 + Random(8);

Draw;

end;

end.