Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Томский государственный педагогический университет»
(ТГПУ)

Д. В. Карташов, Л. А. Непомнящая

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие

Томск
2019

ББК 32.973я73
К 27

Печатается по решению
Учебно-методического совета
Томского государственного
педагогического университета

Карташов, Д. В.
К 27 **Объектно-ориентированное программирование** : учебно-методическое пособие / Д. В. Карташов, Л. А. Непомнящая. – Томск : Изд-во ТГПУ, 2019. – 52 с.

Учебно-методическое пособие содержит основные сведения из теории объектно-ориентированного программирования и примеры решения типовых задач с использованием методов объектно-ориентированного программирования на языке Object Pascal в среде Lazarus.

Учебно-методическое пособие предназначено для преподавания дисциплины «Объектно-ориентированное программирование» обучающимся по направлению подготовки «44.03.05 Педагогическое образование (с двумя профилями подготовки) Направленность (профиль): Математика и Информатика», а также для изучения основ объектно-ориентированного программирования студентами различных направлений, связанных с информатикой и вычислительной техникой.

ББК 32.973я73

Рецензент:
д-р физ.-мат. наук, профессор кафедры информатики
физико-математического факультета Томского государственного
педагогического университета *Л. В. Горчаков*.

© Д. В. Карташов, Л. А. Непомнящая, 2019
© ФГБОУ ВО «ТГПУ», 2019

Содержание

Предисловие ... 4
1. Основные понятия ... 5
  1.1. Объектно-ориентированное программирование (ООП) ... 5
  1.2. Важные понятия и термины ... 6
  1.3. Объект ... 6
  1.4. Класс. Иерархия классов ... 7
  1.5. Инкапсуляция ... 7
  1.6. Наследование ... 7
  1.7. Полиморфизм ... 8
  1.8. Преимущества ООП ... 8
2. Начальные сведения, необходимые для работы с оболочкой Lazarus (Delphi) ... 10
3. Практические задания ... 16
  Задача № 1 «Как дела?» ... 16
  Задача № 2 «a+b» ... 18
  Задача № 3 «Размеры формы» ... 20
  Задача № 4 «10 лет назад» ... 22
  Задача № 5 «Значение функции f(x,y)=(x+y*3) div 2» ... 23
  Задача № 6 «Ювелир» ... 25
  Задача № 7 «Двигающаяся форма» ... 26
  Задача № 8 «Калькулятор» ... 27
  Задача № 9 «График линии» ... 30
  Задача № 10 «График параболы» ... 34
  Задача № 11 «Графический редактор» ... 37
  Задача № 11.1 «Графический редактор. Заливка» ... 42
  Задача № 11.2 «Графический редактор. PaintBox» ... 44
  Задача № 12 «Таймер» ... 45
  Задача № 13 «Светофор» ... 47
Рекомендуемая литература ... 51

Предисловие

В работе изложены основные понятия объектно-ориентированного программирования – объект, класс, свойство, метод. Основные принципы объектно-ориентированного программирования – инкапсуляция, наследование и полиморфизм. Пособие содержит лабораторный практикум по дисциплине «объектно-ориентированное программирование», содержащий задания на создание приложений на языке Object Pascal в среде Lazarus.

Материалы, вошедшие в пособие, апробированы в ходе преподавания дисциплины «Объектно-ориентированное программирование» обучающимися по направлению подготовки «44.03.05 Педагогическое образование (с двумя профилями подготовки) Направленность (профиль): Математика и Информатика» и как часть дисциплины «Технологии программирования» обучающимися по направлению подготовки «09.03.02 Информационные системы и технологии Направленность (профиль): Информационные технологии в образовании».

Пособие может быть использовано при изучении основ объектно-ориентированного программирования студентами различных направлений, связанных с информатикой и вычислительной техникой.

1. Основные понятия

1.1. Объектно-ориентированное программирование (ООП)

Элементы объектно-ориентрованного программирования появились в начале 70-х годов на языке моделирования Симула, а затем получили свое развитие, и в настоящее время объектно-ориентированное программирование принадлежит к числу ведущих технологий программирования.

Основная цель ООП, как и большинства других подходов к программированию, это повышение эффективности разработки программ.

Объектно-ориентированное программирование – это метод программирования, при использовании которого главными элементами программ являются объекты.

Объектно-ориентированное программирование позволяет группировать определенные фрагменты информации вместе с часто используемыми функциями или действиями, связанными с этой информацией, эти элементы объединяются в один, так называемый, объект.

Например:

Группируем музыкальную информацию по «Имя исполнителя» и действию «Воспроизвести все песни этого исполнителя», далее объединяем в объект «Альбом».

В основе ООП лежат 3 понятия:

Инкапсуляция – объединение данных с процедурами и функциями в рамках единого целого – объекта;

Наследование – возможность построения иерархии объектов, с использованием наследования их характеристик;

Полиморфизм – задание одного имени действию, которое передается вверх и вниз по иерархии объектов, с реализацией этого действия способом, соответствующим каждому объекту в иерархии.

Более подробно ознакомимся с ними в следующих параграфах.

Объектно-ориентированное программирование по своей сути – это создание приложений из объектов, похожее на то, как из блоков и разнообразных деталей строятся дома. Одни объекты приходится создавать самостоятельно, тогда, как остальные можно позаимствовать в готовом виде из разнообразных библиотек.

Наиболее распространенными средами разработки для ООП являются: Microsoft Visual Studio, Borland Delphi, Free Pascal Lazarus. Соответственно, наиболее распространенными языками являются – C++, C#, Object Pascal, Java, Visual Basic и т.д.

Концепция объектно-ориентированного программирования позволяет на его основе создавать графические оболочки для быстрого создания графических пользовательских инструментов на основе метода визуального проектирования (Microsoft Visual Studio, Free Pascal Lazarus, Borland Delphi и т.д.).

1.2. Важные понятия и термины

Объект – это совокупность переменных состояний и связанных с ними методов (операций), которые определяют, как объект взаимодействует с окружающей средой.

Класс – обобщенное название набора объектов, обладающих некоторыми одинаковыми методами и структурами данных.

Подкласс – более подробное описание, относящееся к какому-либо специализированному подмножеству набора объектов, описанного классом. Иногда подклассы называют также производными или дочерними классами.

Метод – это функции, относящиеся к конкретному классу, отдельные действия, которые способен будет производить объект.

Атрибут – это характеристика, назначенная элементу (свойство или метод) в определении класса. Атрибуты часто используют для определения будет ли свойство/метод доступен в других частях программы.

Например:

PRIVATE (закрытый) может вызываться только кодом внутри класса, а PUBLIC (открытый) может вызываться любым кодом в программе.

Иерархия классов – это структура многочисленно связанных классов, определяющая, какие классы наследуют функции от других.

1.3. Объект

ООП привносит нам два основных понятия:

– Класс
– Объект

Класс – это абстрактный тип данных, с помощью которого описывается некоторая сущность, ее характеристики и возможные действия. Описав класс, мы можем создать его экземпляр – **объект**, т.е. объект – это конкретный представитель класса.

Например:

Создаем **класс** – преподаватель ТГПУ. У него есть характеристики: факультет, кафедра, стаж и т.п. А Иванов Иван Иванович конкретный представитель данного класса – **объект**.

Объект состоит из нескольких частей:

– Имя объекта;
– Состояние;
– Методы (операции).

Совокупность методов с возможностью управлять состояниями объекта и определять поведение, называется интерфейсом объекта.

1.4. Класс. Иерархия классов

Как правило, все классы имеют одного общего предка. Так как, каждый объект наследует свойства и методы родительского класса, то к объекту можно добавить новые свойства и методы, но нельзя удалить унаследованные.

Таким образом, получаем иерархию классов.

1.5. Инкапсуляция

Инкапсуляция – это свойство объектов скрывать некоторые свои данные и способы их обработки от окружающей их цифровой среды, оставляя «снаружи» только необходимые или требуемые свойства и возможности, т.е. это сокрытие некоторых свойств/возможностей.

Существуют три основных уровня инкапсуляции:

- Private – свойства/возможности видно только в классе.
- Protected – запрещаете использовать вне класса, но разрешаете ее использовать при наследовании
- Public – общедоступный, т.е. разработчик данного класса позволил управлять поведением объектов этого класса.

Применяя инкапсуляцию, мы защищаем данные, принадлежащие объекту, от возможных ошибок, которые могут возникнуть при прямом доступе к этим данным. Также этот принцип может свести к минимуму возможные ошибки и их поиск.

1.6. Наследование

Наследование – метод, в котором объекты могут наследовать свойства и поведение от других объектов, которые называются «родительскими объектами», и добавление чего-то своего (например, свойства).

Например: при изучении курса алгебры вы уже сталкивались с так называемым наследованием, множество натуральных чисел расширяется во множество целых, затем рациональных и т.д. Каждому последующему множеству добавляется определенное свойство/операция. В ООП в качестве «родительского объекта» можно взять простую кнопку серого цвета, при нажатии на которую запускается определенная процедура. На основе этой кнопки можно создать множество кнопок, обладающих различными размерами, цветами и надписями. Нажатие на такую кнопку будет вызывать свою процедуру. Таким образом, все это множество кнопок унаследует свои свойства поведение от «родительского объекта» – простой кнопки.

Преимуществом метода является краткость записи, так как не нужно переписывать одно и то же свойство, а недостатком – сложность создания уникального дизайна.

1.7. Полиморфизм

Полиморфизм – это свойство родственных объектов решать схожие по смыслу проблемы разными способами.

Преимуществами полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного интерфейса для единого класса действий. Выбор конкретного действия возлагается на компилятор.

1.8. Преимущества ООП

От любого метода программирования мы ждем, что он поможет нам в решении наших проблем. Но одной из самых значительных проблем в программировании является сложность. Чем больше и сложнее программа, тем важнее становится разбить ее на небольшие, четко очерченные части. Чтобы побороть сложность, мы должны абстрагироваться от мелких деталей. В этом смысле классы представляют собой весьма удобный инструмент.

- Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
- Данные и операции вместе образуют определенную сущность, и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.
- Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения. Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

Объектно-ориентированное программирование дает возможность создавать расширяемые системы. Это одно из самых значительных достоинств ООП и именно оно отличает данный подход от традиционных методов программирования. Расширяемость означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе выполнения.

Расширение типа и вытекающий из него полиморфизм переменных оказываются полезными преимущественно в следующих ситуациях:

- Обработка разнородных структур данных. Программы могут работать, не утруждая себя, изучением вида объектов. Новые виды могут быть добавлены в любой момент.
- Изменение поведения во время выполнения. На этапе выполнения один объект может быть заменен другим.
- Алгоритмы можно обобщать до такой степени, что они уже смогут работать более чем с одним видом объектов.

– Компоненты не нужно подстраивать под определенное приложение. Их можно сохранять в библиотеке в виде полуфабрикатов и расширять по мере необходимости до различных законченных продуктов.
– Расширение каркаса – независимые от приложения части предметной области могут быть реализованы в виде каркаса и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

К сожалению, многоразового использования программного обеспечения на практике добиться не удается, из-за того, что существующие компоненты уже не отвечают новым требованиям. Однако ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум из многоразового использования компонент. Что даёт нам следующие преимущества:

– Сокращается время на разработку, которое с выгодой может быть отдано другим проектам.
– Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
– Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество работающих с ней программ.
– Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

Другими словами, если у вас, например, стоит задача удалить HTML теги из текста, то вы можете в каждой программе писать эту процедуру заново, а можете вынести её в библиотеку и использовать во всех своих программах, что существенно сократит время на их написание. И если вы обнаружили ошибку в вашем классе, или улучшили его, то это отразится во всех ваших программах, использующих данный класс, без необходимости их переписывания. Более того, вы можете поделиться вашим компонентом с другими людьми или наоборот взять уже готовый.

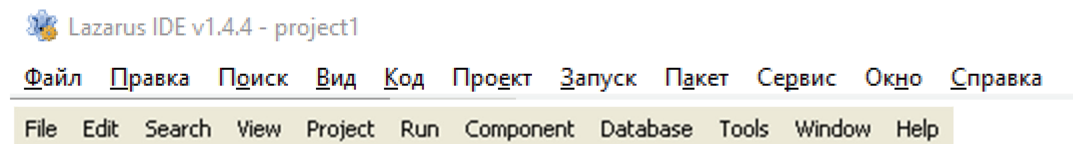
Другим преимуществом является то, что если вы используете готовые библиотеки и поддерживаете их в актуальном состоянии, то повышается надежность и безопасность вашего программного продукта. Ведь мало того, что в данном случае все компоненты протестированы большим количеством людей и на большом варианте программного и аппаратного обеспечения, но и если обнаружится уязвимость, то ее, скорее всего, оперативно исправят.

Вследствие этих преимуществ ООП является в настоящее время самым перспективным, распространенным и эффективным направлением в программировании.

2. Начальные сведения, необходимые для работы с оболочкой Lazarus (Delphi)

После открытия программы Lazarus (Delphi) перед пользователем открывается несколько окон. В целом программа не отличается от стандартных приложений, она также имеет:

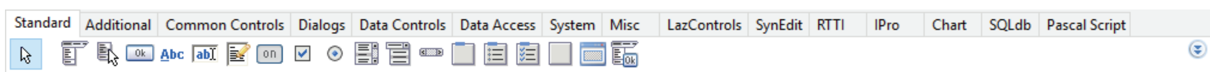
– строку – главное меню



– панель инструментов

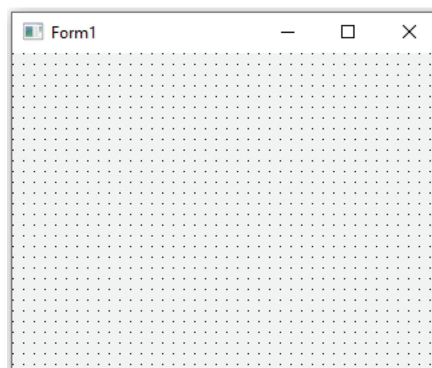
Однако программа имеет и элементы, которых нет в стандартных приложениях:

– Палитра компонент – набор вкладок, на каждой из которых расположены элементы. С помощью этих элементов создаются интерфейсы, элементы называются компонентами, они делятся на визуальные и не визуальные.

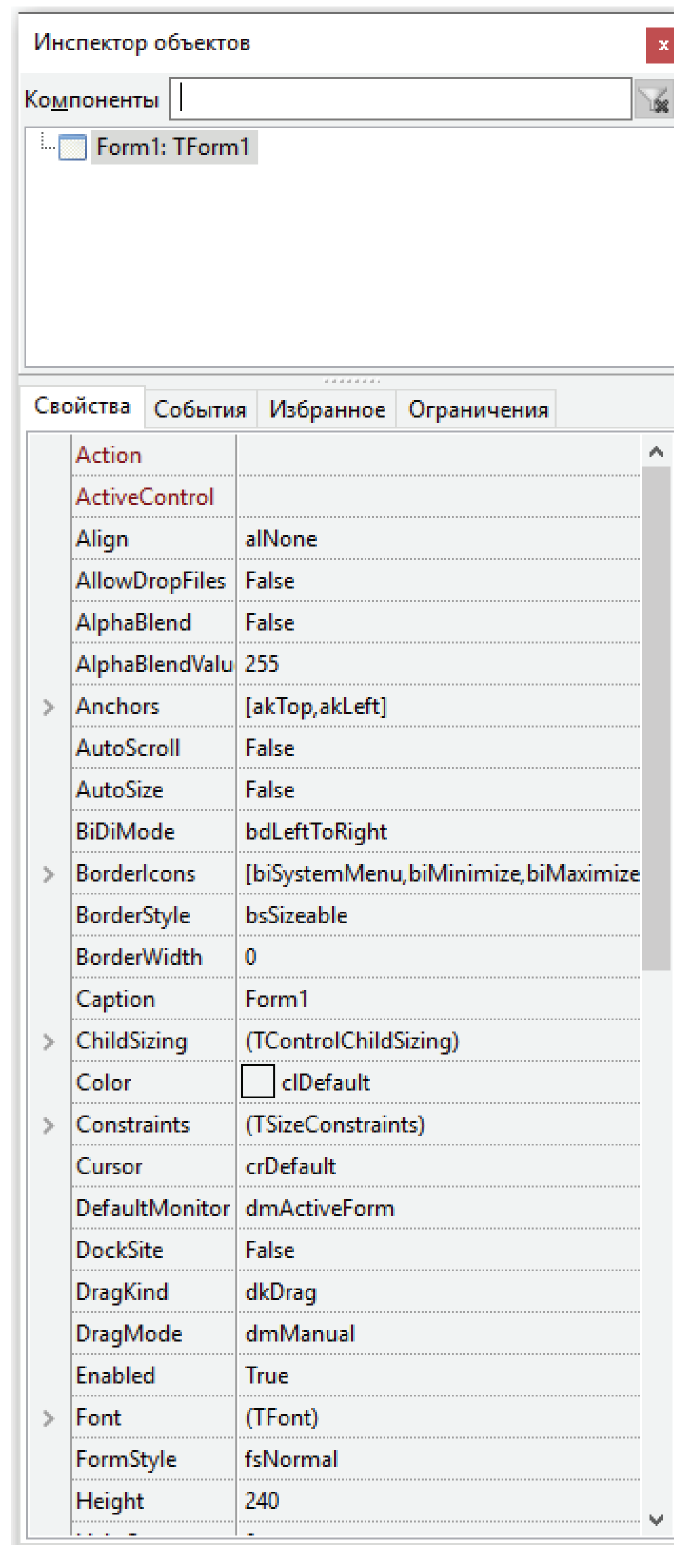


Label1 – служит для вывода текстовой информации
Memo1 – служит для вывода результатов
Edit1 – служит для ввода данных
Button – кнопка

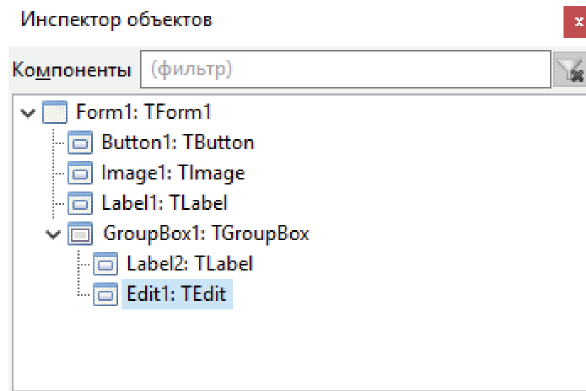
– Дизайнер форм – заготовка окна программы. (Сетку можно отключить)



– Инспектор объектов – окно для настройки свойств (Properties) и событий (Events) выбранного вами объекта.



– Дерево объектов – окно, в котором отображаются все элементы.



– Редактор кода.

Эти окна – основные для работы в Lazarus (Delphi).

Object Pascal

Object Pascal – язык программирования, произошёл от более ранней объектно-ориентированной версии Паскаль.

Изменились группы целых, символьных и строковых типов, их стали разделять на две категории.

– Фундаментальные типы. Их представление в памяти строго фиксируется и остается постоянным во всех последующих реализациях Object Pascal для любых ОС.

– Родовые типы. Их представление в памяти не фиксируется и будет реализовано оптимальным способом, в зависимости от реализации для конкретной ОС.

Объявление класса в Object Pascal – определяет его возможности.

```
type
    ИмяКласса = class (ИмяРодительскогоКласса)
                    <Определение класса>
end;
```

В Object Pascal существуют два метода Create и Free – конструктор и деструктор.

Конструктор – это специальный метод, который создает и инициализирует объект.

Деструкторы – это специальный метод, который разрушает объекты и соответственно, освобождает занимаемую этими объектами память.

Так как Lazarus – визуальная среда разработки, направленная на тех пользователей, которые из готовых объектов создают конкретные приложения, то абсолютно логичным выглядит появление и в Object Pascal новых разделов в описании классов, соответственно.

private – внутренние деталей реализации

protected – интерфейс разработчика

public – run-time интерфейс

published – design-time интерфейс

Все эти разделы работают на уровне модулей, если какая-либо часть объекта доступна в одной области модуля, то такая же доступность будет определена и в другой области.

В Object Pascal добавлена возможность определения полей процедурного типа. Логично, что в теле функций, привязываемых к этим полям, разработчику необходим доступ к другим полям объекта. Возможность такого доступа базируется на передаче в эти функции неявного, но доступного в их коде, параметра, автоматически принимающего значение поля объекта *Self*. Такие функции называются *функциями классов*.

Библиотека визуальных объектов

Рассматривая библиотеку визуальных компонентов, стоит отметить, что все классы произошли от одного общего предка – это класс TObject. Так как, каждый объект наследует свойства и методы родительского класса, приходим к выводу, что к объекту можно добавить новые свойства и методы, но нельзя удалить унаследованные.

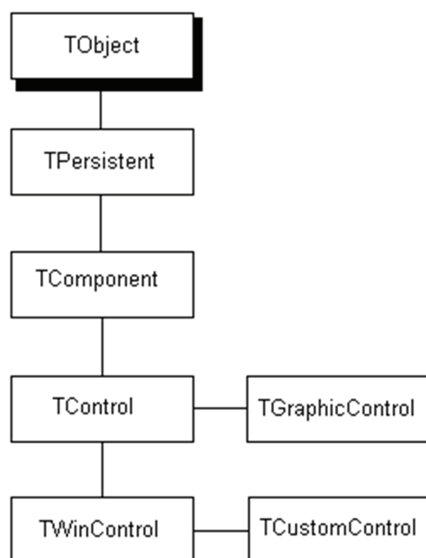


Рис. 1. Классификация объектов

Рассмотрим класс TObject. Он обеспечивает выполнение важнейших функций «жизнедеятельности» любого объекта, его создание и уничтожение. Кроме этого, класс TObject обеспечивает создание и хранение информации об экземпляре объекта и обслуживание очереди сообщений.

Класс TPersistent первый «потомок», обеспечивают возможность взаимодействия объекта с Инспектором объектов, связь объектов друг с другом.

Класс TComponent является важнейшим для всех компонентов, именно он дает возможность создавать не визуальные объекты, обеспечивает взаимодействие компонента со средой разработки, главным образом с Палитрой компонентов и Инспектором объектов.

Следующий класс TControl основное назначение – обеспечить функционирование визуальных компонентов, благодаря ему компоненты научились работать в графическом интерфейсе.

Класс TWinControl расширяет возможности разработчиков по созданию элементов окон управления. (Для создания неоконных элементов управления используется класс TGraphicControl).

На основе класса TWinControl создали еще один класс – TCustomControl, который обеспечивает создаваемые на его основе компоненты с возможностями по использованию канвы.

Класс TCustomControl является общим предком для целой группы классов, обеспечивающих создание различных нестандартных типов оконных элементов управления Windows: редакторов, списков и т.д.

Создание приложений на языке Object Pascal

При работе с компонентами в конструкторе форм Delphi автоматически генерирует соответствующий код на языке Object Pascal, таким образом, для знакомства с ним достаточно начать новый проект.

В Object Pascal исходный код каждой программы разбит на модули.

Модуль состоит из 4 разделов:

Обязательные:

1) Интерфейсный – информация, которая будет доступна из других модулей программы.
2) Раздел реализации – информация, которая из других модулей недоступна.

Не обязательные:

3) Раздел инициализации
4) Раздел завершения.

На основе этих модулей простейшая программа имеет вид.

```
unit название;
interface
{раздел интерфейса}
uses
{список используемых модулей};
const
{список констант};
type
{описание типов};
var
{объявление переменных};
{заголовки процедур, имеющихся в модуле};
{заголовки функций, имеющихся в модуле};
implementation
{раздел реализации}
uses
{список используемых модулей};
const
{список констант};
type
{описание типов};
var
{объявление переменных};
{описание процедур};
{описание функций};
{директивы препроцессора};
Initialization
{раздел инициализации}
{операторы, команды};
finalization
{раздел завершения}
{операторы, команды};
end.
```

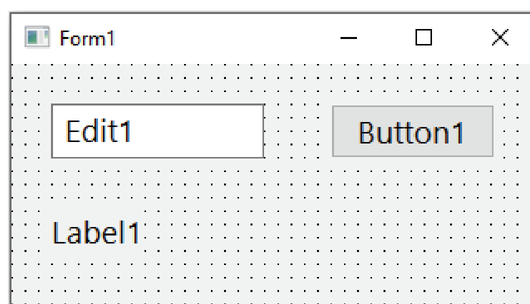
3. Практические задания

Задача № 1 «Как дела?»

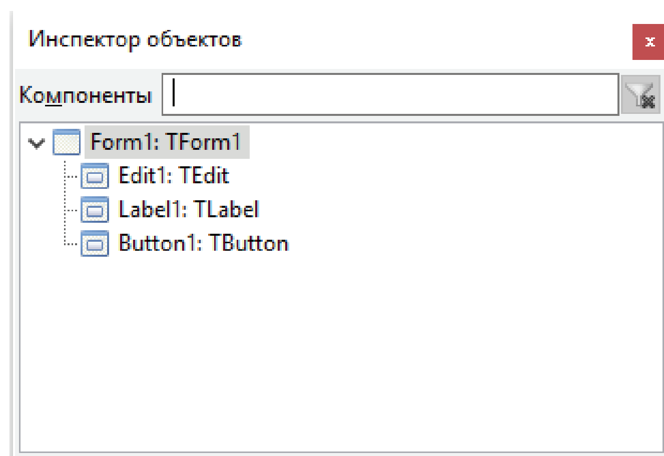
Создайте приложение «Как дела?», которое запрашивает имя пользователя с помощью текстового поля Edit, и выдает личное приветствие с помощью метки Label. Например, если пользователь ввел «Василий», то приложение должно выдать «Привет, Василий! Как дела?».

Создаём новый проект. Нам потребуются следующие объекты:

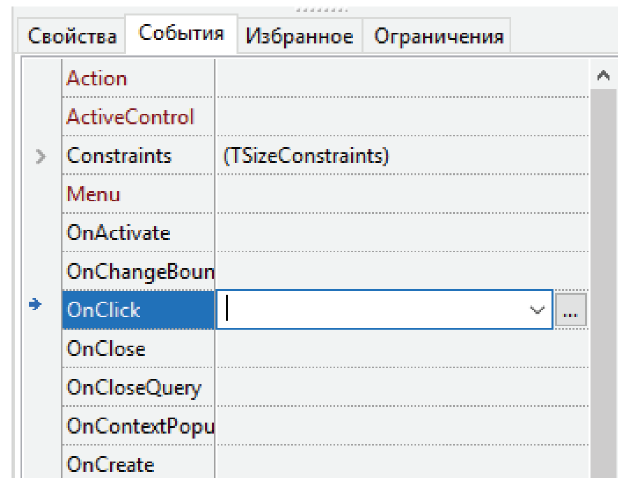
– Edit – 1 шт.
– Label – 1 шт.
– Button – 1 шт.



Теперь вместо «Edit1» напишем подсказку «Введите имя», для этого в свойстве «Text», объекта «Edit1» изменим надпись «Edit1» на «Введите имя». Далее изменим надпись на кнопке «Button1» на «Ок», а «Label1» вообще оставим пустым. Обратите внимание, что у объектов Button и Label нет свойства «Text», поэтому нам нужно изменить у них свойство «Caption». Если вы всё сделали верно, то объект Label как бы исчезнет, но, если вы посмотрите в **инспекторе объектов**, увидите, что он всё еще есть на нашей форме. Почему так получилось? Всё просто – объект Label подстраивается по размеру под введенный в него текст. Можете поэкспериментировать со свойством «Caption» объекта «Label1».



Осталось сделать, чтобы выводилось сообщение «Привет, [Введенное пользователем имя]! Как дела?». Для этого нужно выбрать объект «Button1» и в событиях выбрать «OnClick». Данное событие срабатывает при одном щелчке левой кнопки мыши на кнопке «Button1».



После того, как вы выбрали событие «OnClick», откроется **редактор исходного кода**, а курсор переместится на вновь созданную процедуру

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Теперь давайте сделаем так, чтобы при щелчке по кнопке в метку Label выводился текст «Привет, », Для этого в операторных скобках begin – end напишем следующее:

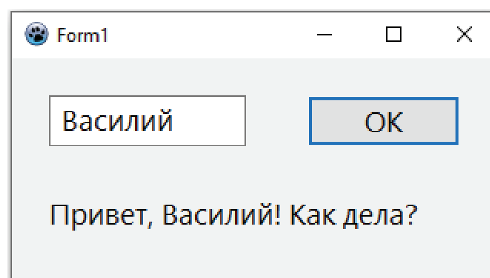
```
Label1.Caption:='Привет, ';
```

Что мы только что сделали? Мы при нажатии на кнопку меняем у объекта «Label1» свойство «Caption» на текст «Привет, ». Давайте запустим программу и убедимся, что это работает.

Теперь нам нужно к данному тексту добавить имя, которое пользователь вводит в поле «Edit». Для этого нам нужно знать, что «склейка текста» в Лазарус производится с помощью знака «+». Соответственно изменяем код.

```
Label1.Caption:='Привет, '+Edit1.text;
```

Осталось поставить «!» знак, после введенного имени и дописать текст «Как дела?».

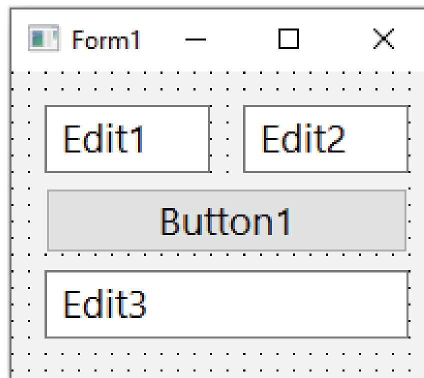


Задача № 2 «a+b»

Создайте приложение «a+b», которое запрашивает два целых числа с помощью текстовых полей Edit, и выдает сумму этих чисел с помощью третьего поля «Edit». Причем третье текстовое поле Edit, должно быть защищено от ввода с клавиатуры.

Создаём новый проект. Нам потребуются следующие объекты:

– Edit – 3 шт.
– Button – 1 шт.



Изменим текст кнопки «Button1» на «=», и удалим текст из объектов Edit, оставив их пустыми. Теперь защитим «Edit3» от ввода с клавиатуры, для этого его свойство «Enabled» изменим на «False». После чего запустим проект. Что изменилось, чем поле «Edit3» отличается от других?

Теперь создадим событие «OnClick» для объекта «Button1», для этого дважды щелкнем по кнопке «Button1» в окне формы. После этого откроется **редактор исходного кода**, а курсор переместится на вновь созданную процедуру

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Почему так получилось? При двойном щелчке по объекту создаётся стандартное, для данного объекта, событие. Для «Button» это «OnClick» – 1 клик ЛКМ по кнопке, для «Edit» это «OnChange» – изменение текста в поле «Edit», для «Form» это «OnCreate» – создание формы (обычно если программа состоит из одной формы, это запуск программы) и т.д.

Теперь нам нужно считать числа, которые вводит пользователь в поля «Edit1» и «Edit2». Сначала нам нужно объявить 2 новые переменные a и b, для этого в разделе VAR, нужно добавить строчку:

a, b: integer;

```
TForm1 = class(TForm)
  Button1: TButton;
  Edit1: TEdit;
  Edit2: TEdit;
  Edit3: TEdit;
private
  { private declarations }
public
  { public declarations }
end;

var
  Form1: TForm1;
  a,b:integer;

implementation

{$R *.lfm}
```

procedure TForm1.Button1Click(Sender: TObject);
begin
a:= Edit1.Text;
end;

Здесь мы в переменную «a» считываем число, которое пользователь ввел в поле «Edit1».

Но если мы сейчас запустим программу на выполнение, то она выдаст ошибку. Почему? Давайте посмотрим, какой тип имеют наши переменные. Переменную «***a***» мы описали как integer, т.е. как целое число, а «**Edit1.Text**», как видно из названия свойства, имеет тип текстовый. Соответственно мы числовой переменной «***a***» присваиваем текстовую информацию. Чтобы исправить данную ошибку нужно значение, введенное в текстовое поле «Edit1», перевести в число. Для этого будем использовать стандартную функцию StrToInt (от String To Integer, т.е. строчный тип в целочисленный). А код изменится:

a:=StrToInt(Edit1.Text);

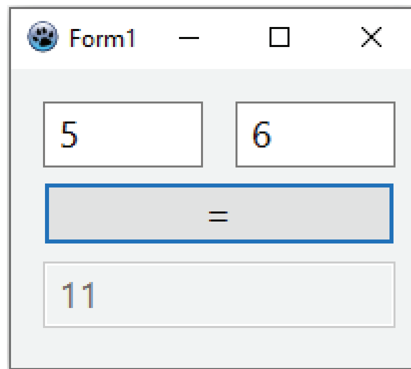
Теперь сделайте то же самое для переменной «b» и поля «Edit2».

Осталось посчитать сумму и вывести результат, в поле «Edit3».

Edit3.Text:=a+b;

Но здесь у нас та же ошибка, только наоборот мы текстовой переменной присваиваем числовую информацию. Можете сами назвать функцию, которая нам понадобится?

Edit3.Text:=IntToStr(a+b);

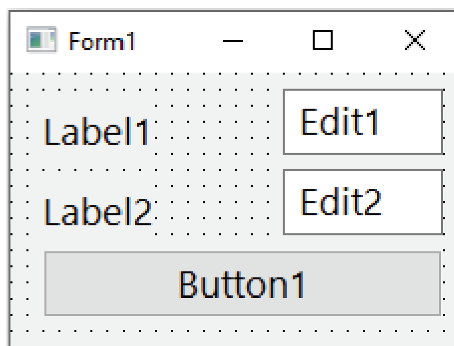


Задача № 3 «Размеры формы»

Создайте приложение с одной кнопкой и двумя областями ввода. В области ввода пользователь вводит размеры окна формы (ширину и высоту). При нажатии на кнопку размеры соответствующим образом изменяются.

Создаём новый проект. Нам потребуются следующие объекты:

- Edit – 2 шт.
- Button – 1 шт.
- Label – 2 шт.



Изменим текст кнопки «Button1» на «Ок», и удалим текст из объектов «Edit», оставив их пустыми. Теперь переименуем «Label»: «Label1» в «Ширина формы:», а «Label2» в «Высота формы:».

Давайте теперь разберемся, как изменить размер формы. Для этого выберем форму и рассмотрим 2 её свойства: «Width» и «Height». Попробуйте поменять их значения, за что они отвечают?

Теперь создадим событие «OnClick» для объекта «Button1», для этого дважды щелкнем по кнопке «Button1» в окне формы и изменим ширину формы, на ту, которую укажет пользователь в поле «Edit1»

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.Width:=StrToInt(Edit1.Text);
end;
```

Почему мы использовали функцию «StrToInt»? Почему свойство **Width** у нас имеет тип integer? Ширина, как и высота формы, у нас задаётся в пикселях, а мы не можем взять половинку пикселя, или другую его часть.

Запустите проект и попробуйте поменять ширину формы. Получилось?

Теперь сделайте то же самое с шириной формы, что у нас изменится в коде **Form1.Width:=StrToInt(Edit1.Text)**? (подсказка: нужно внести 2 изменения).

Теперь мы можем менять не только ширину, но и высоту формы.

Остановите выполнение проекта и запустите его заново, поля «Edit1» и «Edit2» у нас пустые, а это не совсем удобно: во-первых, мы не знаем текущие размеры формы, а во-вторых, если мы сейчас нажмём на кнопку, то программа выдаст ошибку. Почему возникла ошибка? Мы пытаемся преобразовать в число пустое поле – «», что естественно невозможно. Давайте сделаем теперь так, чтобы при запуске программы выводились её размеры. Какое событие нам для этого понадобится?

Выбираем форму и событие «OnCreate». Данное событие будет происходить при запуске нашей программы.

```
procedure TForm1.FormCreate(Sender: TObject);
begin

end;
```

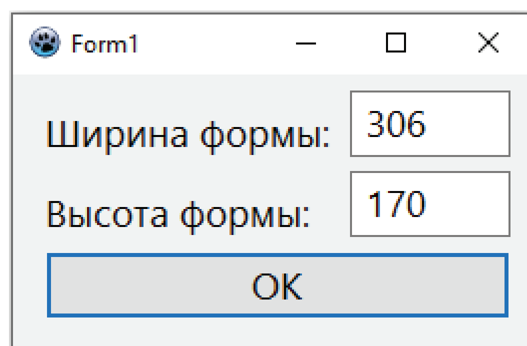
Теперь посмотрим значения свойства «Height» объекта «Form1». У нас, например, оно равно 306 пикселей. Значит, сделаем так, чтобы при запуске программы выводилось число «306» в поле «Edit1».

```
Edit1.Text:='306';
```

Запустите проект и посмотрите, что получилось. Но этот подход не правильный, т.к. если мы захотим изменить размер формы, нам нужно будет менять и число. Поэтому вместо числа «306» будем присваивать высоту формы, так значение у нас будет всегда актуально.

```
Edit1.Text:=IntToStr(Form1. Height);
```

Сделайте то же самое для ширины формы.

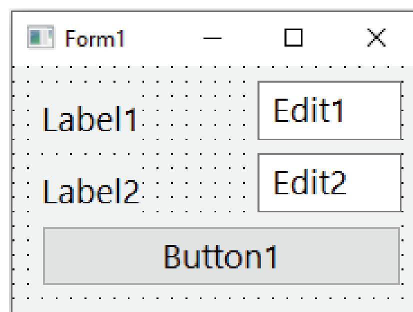


Задача № 4 «10 лет назад»

Создайте приложение «10 лет назад», которое через области ввода запрашивает у пользователя имя и возраст. А при нажатии на кнопку в заголовке формы выводит надпись с именем и возрастом, который был 10 лет назад. Например, ввели имя «Олег», а возраст «21» год, то приложение должно выдать «Олег, 10 лет назад Вам было 11». Причем если было введено число меньше 10, то приложение выводит «Вы еще не родились», а если ровно 10, то «Поздравляю, вы недавно родились» (т.е. человеку было всего несколько месяцев).

Создаём новый проект. Нам потребуются следующие объекты:

- Edit – 2 шт.
- Button – 1 шт.
- Label – 2 шт.



Изменим текст кнопки «Button1» на «Ок», и удалим текст из объектов «Edit», оставив их пустыми. Теперь переименуем «Label»: «Label1» в «Введите имя:», а «Label2» в «Введите возраст:».

Теперь создадим событие «OnClick» для объекта «Button1», для этого дважды щелкнем по кнопке «Button1» и выведем в заголовке формы сообщение: «[Имя], 10 лет назад Вам было [возраст]». Т.е. мы пока будем выводить указанный пользователем возраст, не уменьшая его на 10 лет. Обратите внимание, что нам нужно выводить сообщение в заголовке формы – свойство «Caption».

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.Caption:=Edit1.Text+', 10 лет назад Вам было '+Edit2.Text;
end;
```

Запустите проект и проверьте, что сообщение выдаётся.

Далее нам нужно уменьшить введенный пользователем возраст на 10 лет. Для этого **Edit2.Text** заменим на:

```
IntToStr(StrToInt(Edit2.Text)-10)
```

Попробуйте разобраться, что в данной формуле.

Edit2.Text – это текст, соответственно мы не можем уменьшить его значение на 10, поэтому мы перевели его в число. Теперь из полученного числа

мы можем, вычесть 10, а дальше нам нужно добавить полученное значение к сообщению, поэтому мы переводим его, обратно в текст.

Осталось сделать вторую часть программы – если было введено число меньше 10, то приложение выводит «Вы еще не родились», а если ровно 10, то «Поздравляю, вы недавно родились».

Изменим код:

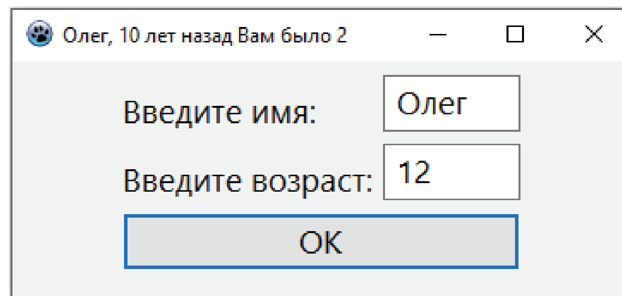
```
if StrToInt(Edit2.Text)<10 then
    Form1.Caption:='Вы еще не родились'
else
    Form1.Caption:=Edit1.Text+', 10 лет назад Вам было '+
    IntToStr(StrToInt(Edit2.Text)-10);
```

Почему мы не можем написать **if Edit2.Text<'10' then…**, ведь оба сравниваемых значения имеют одинаковый формат?

Добавим еще одно условие, на случай если пользователь указал, что его возраст 10 лет.

```
if StrToInt(Edit2.Text)<10 then
    Form1.Caption:='Вы еще не родились'
else if StrToInt(Edit2.Text)=10 then
    Form1.Caption:='Поздравляю, вы недавно родились'
else
    Form1.Caption:=Edit1.Text+', 10 лет назад Вам было '+
    IntToStr(StrToInt(Edit2.Text)-10);
```

Можно ли поменять условия местами? Попробуйте это сделать, не нарушив работу программы.

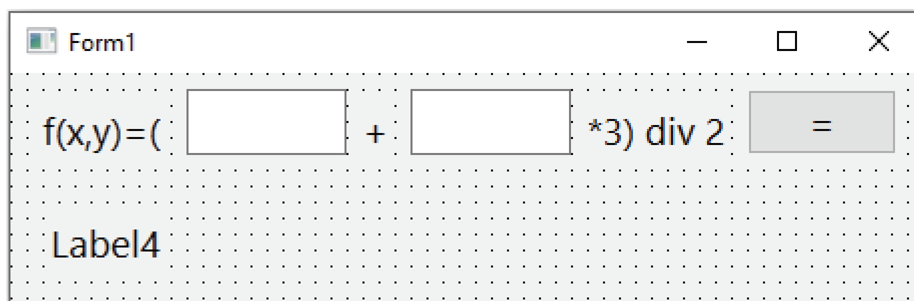


Задача № 5 «Значение функции f(x,y)=(x+y*3) div 2»

Создайте приложение по расчету значения функции f(x,y)=(x+y*3) div 2, где x и y вводятся в полях ввода. Результат отображается в метке. Например, «Значение функции f (x,y) = 35, при x = 5, y = 10». Вводимые значения должны быть целыми.

Создаём новый проект. Нам потребуются следующие объекты:

– Edit – 2 шт.
– Button – 1 шт.
– Label – 4 шт.



В «Label4» будем выводить результат, поэтому её нужно будет очистить.

Создадим событие «OnClick» для объекта «Button1», для этого дважды щелкнем по кнопке «Button1». Теперь нам нужно считать 2 целочисленных переменных.

```
procedure TForm1.Button1Click(Sender: TObject);
var
x,y: integer;
begin

end;
```

Чем данный способ описания переменных отличается от того, что мы использовали ранее? Раньше мы описывали глобальные переменные, доступные в любом месте программы, а в данном случае, переменные у нас будут локальными, т.е. доступны они будут только внутри данной процедуры.

Теперь создадим событие «OnClick» для объекта «Button1» и **считаем значения «Edit1» и «Edit2» в переменные x и y соответственно.**

В поле «Label4» выведите результат формулы (x+y*3) div 2.

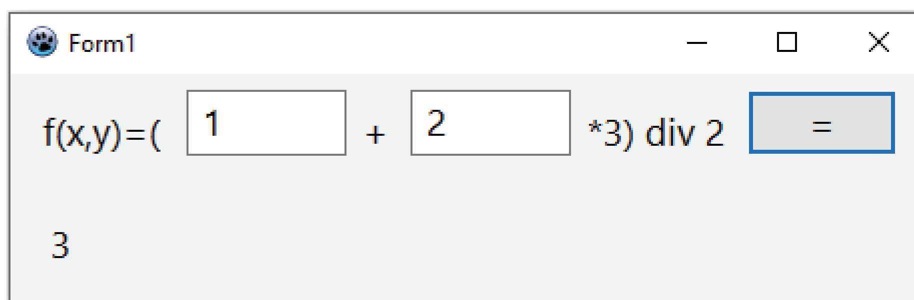
Что делает оператор div? В какой последовательности будут выполняться операции?

Div – это деление нацело. Например, если 7 разделить на 3, то получим

7:3=3*2+1

т.е. 2 целых и 1 в остатке. Соответственно если выполнить div, то получим 2, а если mod, то остаток от деления – 1.

Приоритет операций у нас стандартный, т.е. div у нас имеет такой же приоритет, как и обычная операция деления. Сначала выполнится операции в скобках: умножения и вычитания, а лишь потом деления нацело.

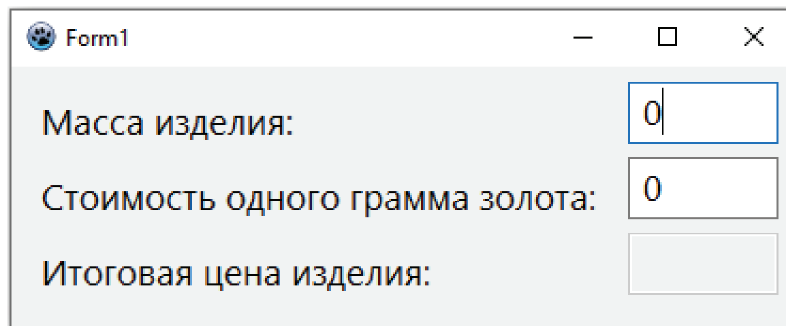


Задача № 6 «Ювелир»

Создайте приложение «Ювелир», вычисляющее стоимость золотого изделия. В однострочных редакторах пользователь заносит массу изделия и стоимость (в руб.) одного грамма золота. Итоговая цена изделия должна отобразиться в отдельном однострочном редакторе Edit. Учтите, что масса может быть не целой величиной.

Создаём новый проект. Нам потребуются следующие объекты:

- Edit – 3 шт.
- Label – 3 шт.



Для начала нам нужно объявить 3 переменные:

```
s: integer;
m, ic: real;
```

где ***s*** – стоимость (в руб.) одного грамма золота, будем считать, что она будет целой, а ***m*** и ***ic*** масса изделия и итоговая цена изделия, соответственно.

Давайте сделаем так, чтобы при изменении «Edit1» или «Edit2» автоматически будет пересчитываться значение «Edit3». Выберите «Edit1», в окне «Инспектор объектов» во вкладке «События» выберите «OnChange», теперь при изменении значения, будет происходить событие «Edit1Change».

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
    m:=StrToFloat(Edit1.Text);
    s:=StrToInt(Edit2.Text);
end;
```

Как вы заметили у нас появилась новая функция **StrToFloat**, как вы думаете, что она делает?

Теперь перемножим массу на стоимость 1 грамма золота, и выведем итоговую стоимость изделия в поле «Edit3».

```
ic:=m*s;
Edit3.Text:=FloatToStr(ic);
```

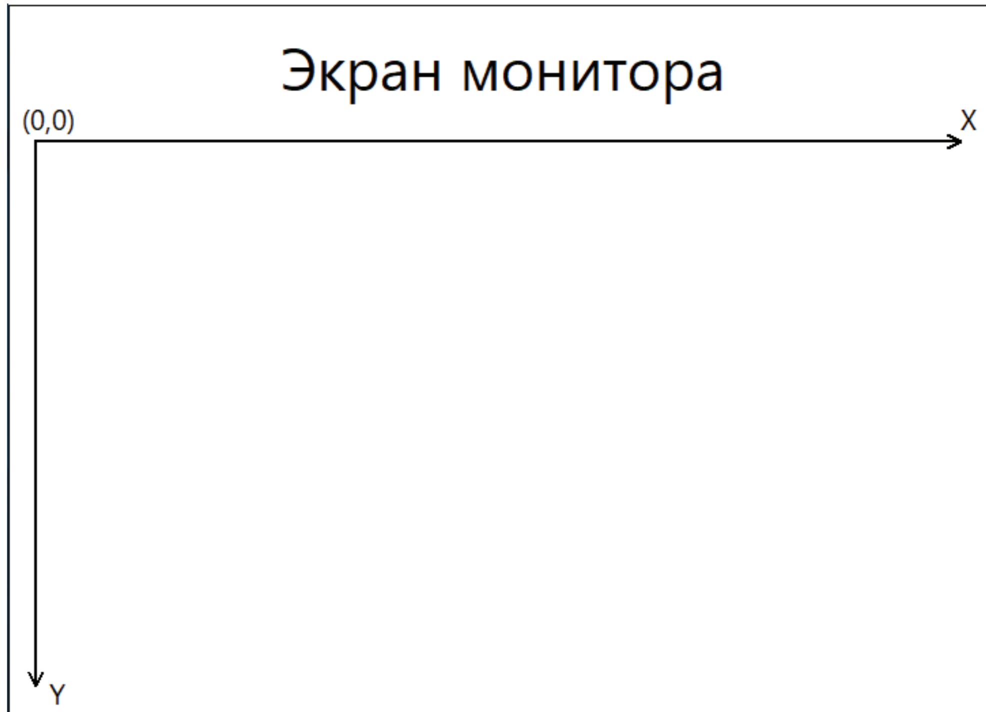
Запустите программу и посмотрите, пересчитывается значение «Edit3» при изменении «Edit1». (Если при любых значениях получается 0, то измените значение «Edit2» на любое отличное от 0).

Теперь нам нужно сделать так, чтобы итоговая стоимость считалась не только при изменении значения «Edit1», но и «Edit2». Для этого выберите «Edit2», в окне «Инспектор объектов» во вкладке «События» выберите «OnChange» и в выпадающем списке укажите уже созданное ранее событие «Edit1Change». Теперь при возникновении 2 разных событий, будет выполняться одно и тоже действие. Что это *инкапсуляция, наследование* или *полиморфизм*?

Задача № 7 «Двигающаяся форма»

Создайте приложение «Двигающаяся форма» с четырьмя кнопками. При нажатии на эти кнопки форма перемещается на 10 пикселей влево, вправо, вверх либо вниз соответственно. В заголовке формы отображаются текущие координаты формы. Например, «Форма имеет координаты: x=95, y=124».

Для начала давайте определимся, что называется текущими координатами формы – это координаты левого, верхнего угла формы, относительно левого, верхнего угла экрана, благодаря им можно однозначно указать положение формы на экране монитора. При этом, если ось OX располагается также как в декартовой системе координат, слева направо, то ось OY направлена сверху вниз.



Создаём новый проект. Нам потребуются следующие объекты:
– Button – 4 шт.

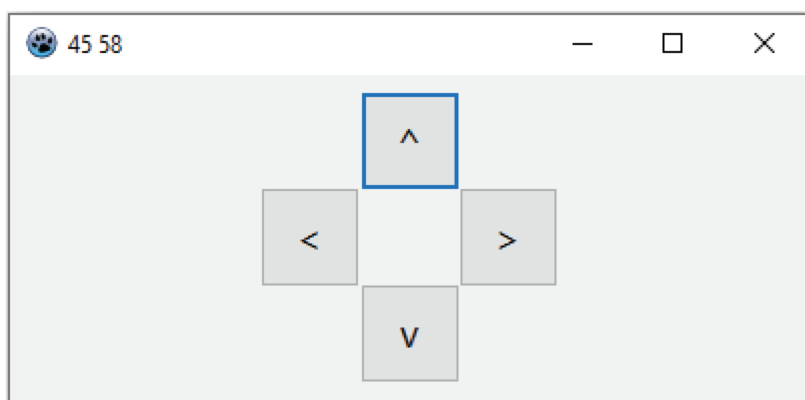
Начнём с кнопки «вверх», чтобы форма поднялась на 10 пикселей вверх, нужно уменьшить расстояние от верхнего края экрана до формы на 10 пикселей. Т.е. уменьшить значение свойства формы **top**. А чтобы форма сдвинулась влево нужно уменьшить свойство формы **left**.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Form1.Top:=Form1.Top-10;
end;
```

Самостоятельно сделайте остальные 3 кнопки.

Теперь сделаем так, чтобы в заголовке формы выводились её координаты, для этого ко всем кнопкам, в конце, добавим следующую строчку.

```
Form1.Caption:=IntToStr(form1.Left)+' '+ IntToStr(form1.Top);
```

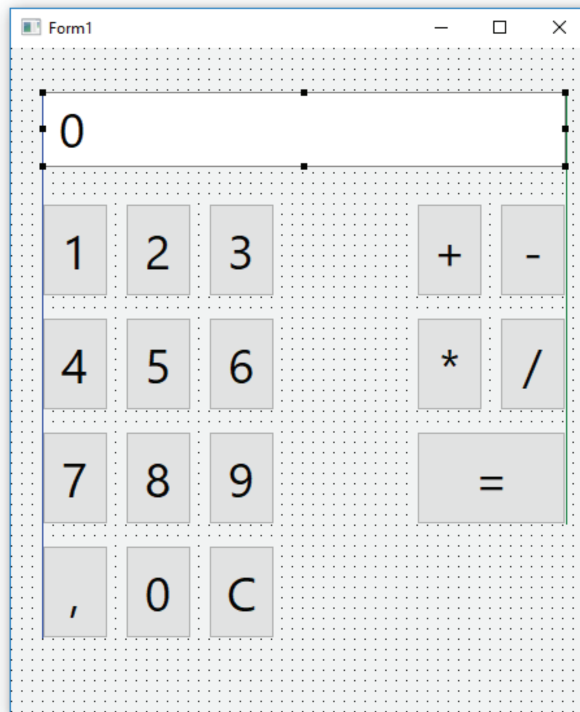


Самостоятельно сделайте так, чтобы координаты выводились в виде фразы: «Форма имеет координаты: x=45, y=58».

Задача № 8 «Калькулятор»

Создаём новый проект. Нам потребуются следующие объекты:

- Edit – 1 шт.
- Button – 17 шт.



Теперь выберите объект Edit1 и задайте ему следующие свойства:

- Alignment – taRightJustify
- Enabled – False
- Text – «»

За что отвечают эти свойства?

Теперь напишем код для кнопки «1»

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Edit1.Text:='1';
end;
```

Почему работает неправильно? Модернизируйте код:

```
Edit1.Text:=Edit1.Text+'1';
```

Самостоятельно сделайте это для остальных кнопок. Как должны работать кнопки «,» и «С» (сброс – очистка окна ввода).

Теперь откройте любое приложение «калькулятор» (на компьютере или телефоне) и посмотрите, что происходит при нажатии на кнопку «+». Давайте перечислим эти действия:

- Запоминаем выбранное действие (в нашем примере – операция сложения);
- Запоминаем введенное число;
- Очищаем окно ввода.

Следовательно, нам потребуется завести следующие переменные:

- a, b: real; – для хранения первого и второго введенных чисел;
- znak: char; – для хранения информации об выбранном действии.

Соответственно код будет выглядеть так:

```
procedure TForm1.Button11Click(Sender: TObject);
begin
    znak:='+';
    a:=strtofloat(edit1.text);
    edit1.Text:='';
end;
```

Сделайте то же самое для остальных кнопок.

Теперь посмотрите, что происходит при нажатии на «=».

Давайте перечислим эти действия:

- Запоминаем введенное число;
- Если была выбрана операция сложения, то выводим сумму и т.п.

Код будет выглядеть так:

```
procedure TForm1.Button15Click(Sender: TObject);
begin
    b:=strtofloat(edit1.Text);
    if znak='+' then c:=a+b;
    edit1.Text:=floattostr(c);
end;
```

Сделайте аналогично, для остальных знаков.

Запустите приложение и введите следующий пример: 2/0, что произойдет? Как этого можно избежать?

Давайте заменим строку:

```
edit1.Text:=floattostr(c);
```

На следующее, где «oshibka» это текстовая переменная содержащая текст ошибки:

```
if oshibka<>'' then begin
    edit1.Text:=oshibka;
    oshibka:=''
end else edit1.Text:=floattostr(c);
```

Т.е., если переменная «oshibka» не пуста, то выводим текст ошибки, иначе выводим результат выполнения операции.

Исправили ли мы ошибку из нашего примера 2/0? Почему?

Исправим код для знака «/»:

```
if znak='/' then begin
    if b=0 then oshibka:='Деление на 0'
    else c:=a/b;
end;
```

На данный момент, чтобы после вычисления нам ввести новое число, нам нужно очистить окно ввода. Давайте сделаем это автоматически. Если воспользоваться способом, который мы использовали на кнопке – знаков операций (edit1.Text:='';), то мы не увидим результат, т.к. он тут же затрется.

Поэтому нам нужно различать 2 ввода чисел:

- Добавление цифры к уже введенному числу (что мы уже сделали);
- Очистить окно ввода и начать вводить новое число.

Второе событие происходит только после того, как была нажата кнопка «=», поэтому нам нужна переменная, которая будет отслеживать нажатие данной кнопки, назовем её «if_ravno».

Добавьте на кнопку «=» следующий код:

```
if_ravno:=true;
```

Теперь заменим код для кнопки «1»:

```
if if_ravno then begin
    Edit1.Text:='1';
    if_ravno:=false
end else Edit1.Text:=Edit1.Text+'1';
```

Повторите это изменение для других кнопок.

Чему равны переменные oshibka if_ravno при запуске программы? Создайте новое событие и пропишите туда следующее:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    oshibka:='';
    if_ravno:=false;
end;
```

Задание для самостоятельной работы

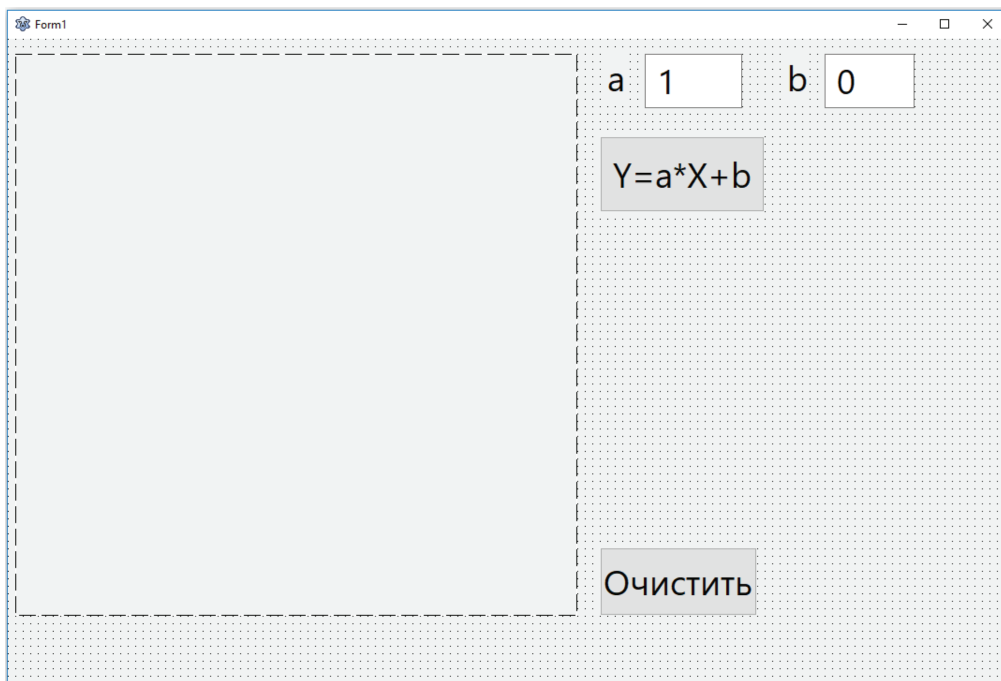
1. Мы сделали калькулятор для бинарных операций, работающих с двумя числами. Создайте кнопку для унарной операции, например, для возведения числа в квадрат, извлечение квадратного корня и т.п.

2. По желанию модернизируйте кнопки операций, по аналогии с кнопкой «=», т.е., чтобы после нажатия, например, на кнопку «+» введенное число не стиралось, а также как на кнопке «=» следующее число вводилось бы с нуля.

Задача № 9 «График линии»

Создаём новый проект. Нам потребуются следующие объекты:

– Edit – 2 шт.
– Label – 1 шт.
– Button – 2 шт.
– Image – 1 шт.

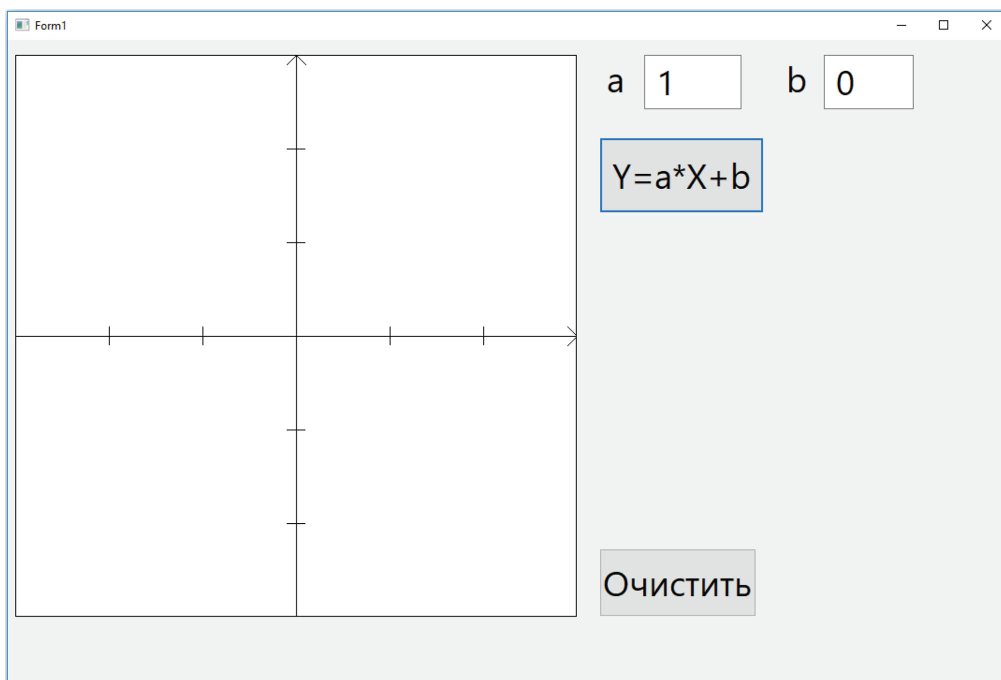


Компонент Image предназначен для отображения на форме графических изображений, по умолчанию выводит на поверхность формы изображения, представленные в bmp формате.

Теперь выберите объект Image1 и задайте ему следующие свойства:

– Height – 600
– Width – 600

Далее необходимо подготовить форму для рисования графика, для этого нарисуем оси координат и укажем единичные отрезки. За единичный отрезок возьмем 100 пикселей.



Для начала нам нужно подготовить объект Image1: выбрать цвет пера по умолчанию чёрный, а само поле залить белым цветом.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Image1.Canvas.Pen.Color:=clBlack;
    Image1.Canvas.Brush.Color:=clWhite;
    Image1.Canvas.Rectangle(0,0,600,600);
end;
```

Свойство **Canvas** это холст, который позволяет программисту иметь доступ к каждой своей точке (пикселю), и словно художнику отображать то, что требуется. Конечно, рисовать попиксельно для работы с графикой в Delphi не приходится, система Lazarus предоставляет для этого мощные средства работы с графикой, облегчающие задачу программиста.

В работе с графикой в Lazarus в распоряжении программиста находятся канва (холст, полотно – свойство Canvas Lazarus компонентов), карандаш (свойство Pen), кисть (свойство Brush) того компонента или объекта, на котором предполагается рисовать. У карандаша Pen и кисти Brush можно менять цвет (свойство Color) и стиль (свойство Style). Доступ к шрифтам предоставляет свойство канвы Font. Эти инструменты позволяют отображать как текст, так и достаточно сложные графики математического и инженерного содержания, а также рисунки. Кроме этого, работа с графикой позволяет использовать в Lazarus такие ресурсы Windows, как графические и видеофайлы.

Строка **Image1.Canvas.Pen.Color:=clBlack;** задаёт цвет карандаша чёрный (на скриншоте это цвет обводки). Попробуйте его изменить на **clRed**.

Строка **Image1.Canvas.Brush.Color:=clWhite;** задаёт цвет кисти (заливки), которым закрашиваются объекты, у которых есть внутренняя область, в нашем случае прямоугольник **Rectangle.**

Строка **Image1.Canvas.Rectangle(0,0,600,600);** рисует прямоугольник, который совпадает по размерам с объектом **Image1.** Но если мы изменим размеры **Image1**, то прямоугольник не будет с ним совпадать, как мы уже знаем верхняя левая точка всегда будет иметь координаты (0,0), нижняя правая, в нашем случае (600,600).

Изменим первую цифру 600 на

Image1.width

Тоже самое сделайте со второй цифрой. Что будет вместо **width**?

Нарисуем ось OX

```
Image1.Canvas.Line(0, Image1.Height div 2,
    Image1.width, Image1.Height div 2);
```

Добавим единичный отрезок на ось OX

```
Image1.Canvas.Line(100,290,100,310);
```

Нарисуем стрелочку

```
Image1.Canvas.Line(590,290,600,300);
Image1.Canvas.Line(590,310,600,300);
```

Самостоятельно нарисуйте ось OY, все единичные отрезки, и направление оси OY.

Теперь у нас возникает проблема, что строить график мы будем в нашей новой системе координат XOY, которая не совпадает с системой координат Lazarus – $X_{Laz}O_{Laz}Y_{Laz}$, причем, оси абсцисс и оси ординат обеих систем параллельны, но оси ординат разно направлены. Из курса геометрии вы должны помнить формулу параллельного переноса осей координат:

X_{Laz}=X+(смещение по оси абсцисс)

Но так как оси OY и $O_{Laz}Y_{Laz}$ разно направленны, то у одной из них должен измениться знак при переводе, поэтому вторая формула будет выглядеть так:

Y_{Laz}=-Y+(смещение по оси ординат)

Осталось устранить вторую проблему – в наших системах координат разные единичные отрезки, так в системе $X_{Laz}O_{Laz}Y_{Laz}$ – единичный отрезок – 1 пиксель, а в системе XOY – произвольный отрезок, в нашем случае 100 пикселей.

Поэтому итоговая формула будет выглядеть так:

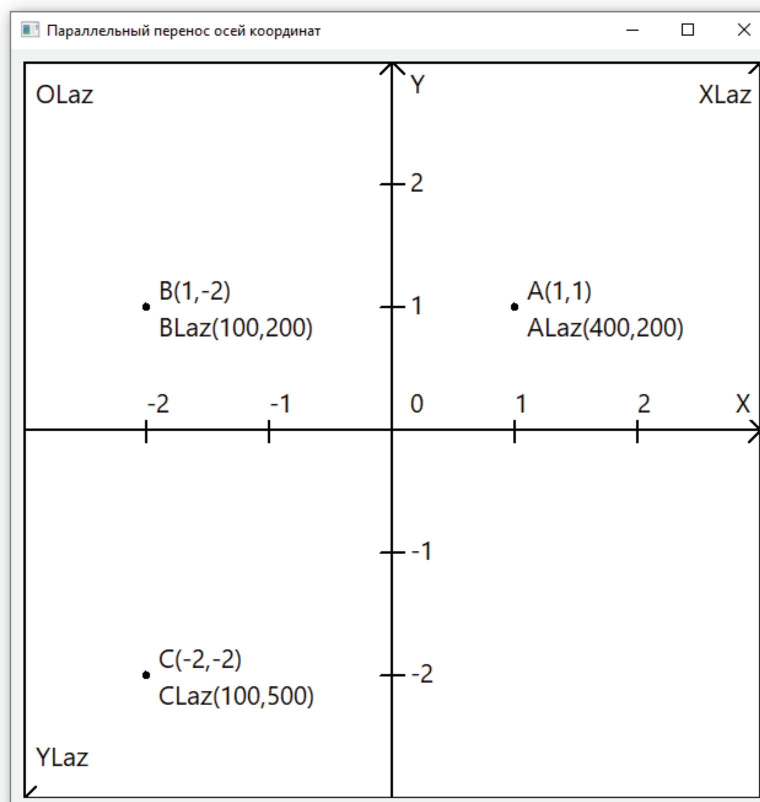
X_{Laz}=X*(ед. отрезок) + (смещение по оси абсцисс (X))

Y_{Laz}=-Y*(ед. отрезок) + (смещение по оси ординат (Y))

В нашем случае:

$X_{Lazarus}$=X*100 + 300

$Y_{Lazarus}$=-Y*100 + 300



Проверим нашу формулу на точке А

$X_{Lazarus}$=X*100 + 300 = 1*100 + 300 = 100 + 300 = 400

$Y_{Lazarus}$=-Y*100 + 300 = -(1)*100 + 300 = -100 + 300 = 200

Самостоятельно проверьте точки B и C.

Для начала нарисуем формулу прямой **y=x**:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    x:=-2;
    y:=x;
    Image1.Canvas.MoveTo(300+x*100,300-y*100);
    x:=2;
    y:=x;
    Image1.Canvas.LineTo(300+x*100,300-y*100);
end;
```

Запустите программу и посмотрите, как выглядит график. Видно, что вместо прямой мы рисуем отрезок от X равного -2 до +2.

Код выше состоит из двух частей, считаем, что при X = -2, Y будет равно X, то есть -2, ставим курсор в данную точку (свойство **MoveTo**).

Далее считаем вторую точку, куда будет идти прямая, и рисуем линию от точки, где сейчас находится курсор, до нашей новой точки (свойство **LineTo**). Теперь курсор находится в точке **(300+x*100,300-y*100)** и следующая линия будет рисоваться от конца этой линии, если мы не переместим курсор в новую точку.

Самостоятельно заведите 3 переменные: SmeshhenieX, SmeshhenieY, EdinichnyjOtrezok в разделе констант и присвойте им значение 300, 300 и 100 соответственно, за что отвечают эти переменные?

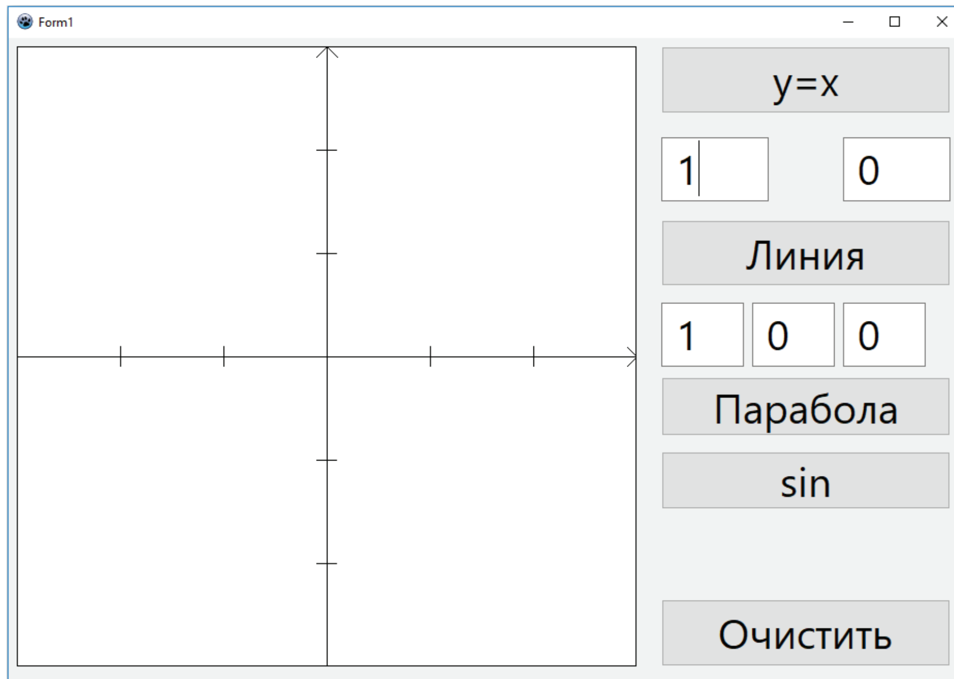
Замените формулы, из кода выше, используя эти переменные.

Переделайте программу для графика y=a*x+b.

Сделайте кнопку «Очистить», т.е. вернуться к начальному состоянию окна, соответствующему событию FormCreate.

Задача № 10 «График параболы»

Измените предыдущий проект:

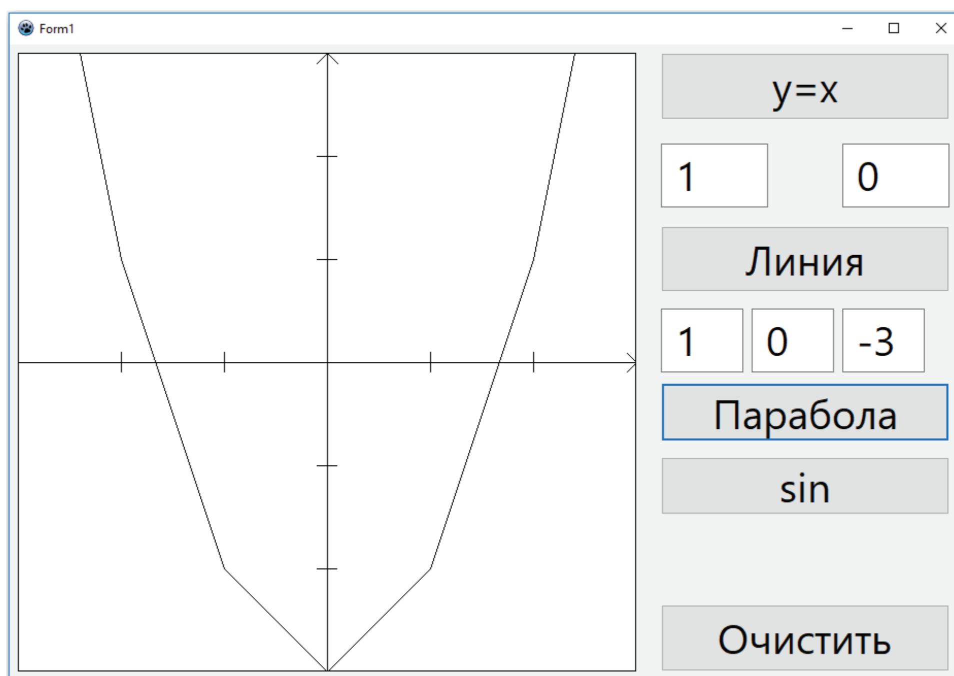


Добавим возможность строить график параболы и синуса.

В отличие от прямой, которую можно нарисовать по двум точкам, параболу рисовать сложнее, мы будем рисовать её в цикле:

1. берем x = -3 и считаем в этой точке значение y;
2. ставим в эту точку курсор;
3. сдвинем x на 10 пикселей вправо (т.е. добавим к x 0.1 в нашей системе координат) и посчитаем значение y;
4. нарисуем линию от первой точки до второй;
5. повторяем шаги 3–4 до тех пор, пока x не станет равен 3.

Пример параболы при шаге в 100 пикселей:



Начало программы «парабола» очень похоже на начало программы «линия»:

```
x:=-3;
y:=sqr(x);
Image1.Canvas.MoveTo(SmeshhenieX + round(x*EdinichnyjOtrezok),
SmeshhenieY - round(y*EdinichnyjOtrezok));
```

Зачем нужно округление **round**?

Поскольку мы используем уже дробные числа, то может возникнуть ситуация, когда наша точка будет приходиться на часть пикселя, например точка (2,555; 6,528025) = $(255,5; 652,8025)_{Lazarus}$, а пиксель нельзя поделить на кусочки и взять половинку.

Соответственно, x и y теперь не могут быть integer и сменят тип на real.

Теперь добавим цикл, поскольку в 5 пункте условие стоит «пока», логичнее всего использовать цикл **while**.

```
while x<=3 do begin
    x:=x+0.1;
    y:=sqr(x);
    Image1.Canvas.LineTo(SmeshhenieX + round(x*EdinichnyjOtrezok),
    SmeshhenieY - round(y*EdinichnyjOtrezok));
end;
```

Парабола готова.

Переделайте программу под график $y=a*x^2+b*x+c$

Сделайте график функции y=sin(x)

Измените, единичный отрезок со 100 пикселей до 10, нарисуйте на осях координат единичные отрезки и поправьте граничные интервалы в графиках.

Подсказка, единичные отрезки нужно рисовать в цикле, т.е. их будет по оси OX Image1.width div (единичный отрезок), в нашем случае 600 div 10 = 60 (на самом деле 59, т.к. 60 отметка совпадет с границей Image) и рисовать каждый вручную, как делали раньше не вариант.

Алгоритм такой же, как и с графиком параболы:

1. считаем x = 0;
2. увеличиваем x на 10 пикселей;
3. рисуем единичный отрезок;
4. повторяем шаги 2–3 59 раз.

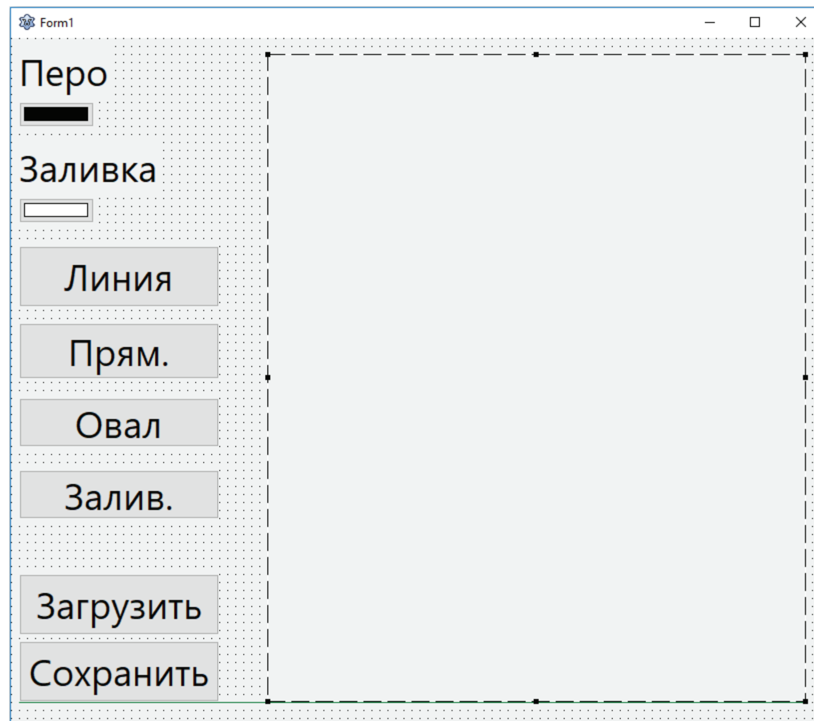
```
for i:=1 to (Image1.Width div EdinichnyjOtrezok)-1 do
    Image1.Canvas.Line(i*EdinichnyjOtrezok, (Image1.Height div 2)-5,
    i*EdinichnyjOtrezok, (Image1.Height div 2)+5);
```

Аналогично, для оси OY.

Задача № 11 «Графический редактор»

Создаём новый проект. Нам потребуются следующие объекты:

- Label – 2 шт.
- ColorButton – 2 шт. (вкладка Misc)
- Button – 6 шт.
- Image – 1 шт.



Компонент **Image** дает отображение на форме графического изображения. Свойство **Picture** типа **TPicture** содержит отображаемую графическую составляющую, у которой тип – битовая матрица, пиктограммы, метафайла или определенного пользователем типа. Свойство **Canvas** позволяет создавать и редактировать изображения.

Во время проектирования загрузить в свойство **Picture** графический файл можно щелкнув на кнопке с многоточием около свойства **Picture** в окне Инспектора Объектов. Должно открыться окно Picture Editor, которое позволит загрузить в свойство **Picture** некоторый графический файл (через кнопку Load), и сохранить открытый файл, дав ему новое имя или сохранить в новом каталоге.

Когда вы в процессе создания проекта сделали загрузку изображения из файла в компонент **Image**, он не просто отобразит его, но и сохранит загруженное изображение в приложении. Преимущества состоят в том, что ваш проект в целом будет поставляться без отдельного графического файла.

При установке свойства **AutoSize** равному **true**, размер компонента **Image** будет автоматически подогнан под размер помещенной в **Image** картинки. Если свойство **AutoSize** установлено в значение **false**, то изображение может

и не уместиться в компонент или же, наоборот, площадь компонента возможно окажется намного больше площади изображения.

Другое свойство – **Stretch** позволяет реализовать подгонку не компонента под размер рисунка, а сам рисунок под параметр компонента. Но поскольку вряд ли реально в действительности установить размеры **Image** очень точно пропорциональными величине рисунка, то изображение будет искажено. При установке **Stretch** равное **true** может иметь смысл только для каких-то узоров, но не для картинок. Свойство **Stretch** не действует на изображения пиктограмм, которые не меняют своих размеров.

Свойство **Center**, установленное в **true**, размещает изображение в центре **Image**, если параметры размера компонента больше параметра размеров рисунка.

Свойство – **Transparent** – прозрачность. Если значение **Transparent** равно **true**, то изображение в **Image** будет прозрачным. Это используется при наложении изображений друг на друга. Следует помнить, что свойство **Transparent** распространяется только на битовые матрицы. При этом прозрачным (т.е. заменяемым на цвет расположенного под ним изображения) делается по умолчанию цвет левого нижнего пиксела битовой матрицы.

Свойство **Picture** позволяет легко организовать обмен с графическими файлами любых типов в процессе выполнения приложения. Это свойство – объект, который имеет в свою очередь подсвойства, которые указывают на хранящийся графический объект. Если в **Picture** хранится битовая матрица, на нее указывает свойство **Picture.Bitmap**. Если хранится пиктограмма, на нее указывает свойство **Picture.Icon**. На хранящийся метафайл указывает свойство **Picture.Metafile**. Наконец, на графический объект произвольного типа указывает свойство **Picture.Graphic.**

Объект **Picture** и его свойства **Bitmap, Icon, Metafile** и **Graphic** имеют методы файлового чтения и записи **LoadFromFile** и **SaveToFile**. Для свойств **Picture.Bitmap, Picture.Icon** и **Picture.Metafile** формат файла должен соответствовать классу объекта: битовой матрице, пиктограмме, метафайлу. При чтении файла в свойство **Picture.Graphic** файл должен иметь формат метафайла. А для самого объекта **Picture** методы чтения и записи автоматически подстраиваются под тип файла.

Теперь выберите объект **Image1** и задайте ему следующие свойства:
- AutoSize – true (чтобы **Image1** подстраивался под размер загруженной картинки)

У объекта **ColorButton1** задайте следующие свойства:
- ButtonColor – clBlack (цвет по умолчанию – чёрный (можете выбрать любой))

У объекта **ColorButton2** задайте следующие свойства:
- ButtonColor – clWhite (цвет по умолчанию – белый (можете выбрать любой))

Теперь подготовим объект **Image1** к рисованию, зальём его белым цветом, а рамку сделаем чёрной. (см. задачу № 9)

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Image1.Canvas.Brush.Color:=clWhite;
    Image1.Canvas.Pen.Color:=clBlack;
    Image1.Canvas.Rectangle(0,0,Image1.Width,Image1.Height);
end;
```

Сделаем кнопку, отвечающую за рисование линии:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
f:=1;
end;
```

Также, как и у калькулятора она отвечает, что выбрано действие – рисование линии **f:=1**. Соответственно в **procedure TForm1.FormCreate** нужно добавить строчку

f:=0;

Означающую, что у нас изначально ничего не выбрано.

Как будем рисовать линию:

1. При нажатии ЛКМ мы запоминаем координаты, пикселя, на который указывает курсор. Событие **MouseDown.**
2. Когда мы отпускаем ЛКМ мы рисуем линию, от запомненных в шаге 1 координат, до точки в которой мы ЛКМ отпустили. Событие **MouseUp.**

```
procedure TForm1.Image1MouseDown(Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    DownX:=X;
    DownY:=Y;
end;
```

Если посмотреть внимательней, то у события **MouseDown**, в скобках, есть две переменные **X, Y**, это и есть координаты события нажатия ЛКМ (не путать с щелчком мыши, при котором происходит 2 события: нажали ЛКМ и отпустили её). Но т.к. эти переменные объявлены внутри процедуры, то и использовать их мы можем внутри неё, поэтому нужно сделать их глобальными: **DownX** и **DownY**.

Теперь напишем код для второго события:

```
procedure TForm1.Image1MouseUp(Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    Image1.Canvas.Pen.Color:=ColorButton1.ButtonColor;
    Image1.Canvas.Brush.Color:=ColorButton2.ButtonColor;
    if f=1 then
        Image1.Canvas.Line(DownX,DownY,X,Y);
end;
```

Первое, что нужно сделать это узнать цвет пера (**Pen.Color**) и цвет кисти / заливки (**Brush.Color**), а потом если выбрана линия, то нарисовать её.

Сделайте по аналогии с линией (*Line*) кнопки для рисования прямоугольника (*Rectangle*) и эллипса (*Ellipse*).

У прямоугольника – ***Rectangle*** стороны расположены параллельно осям координат, поэтому для его задания достаточно двух точек (как и для линии): верхней-левой и нижней-правой.

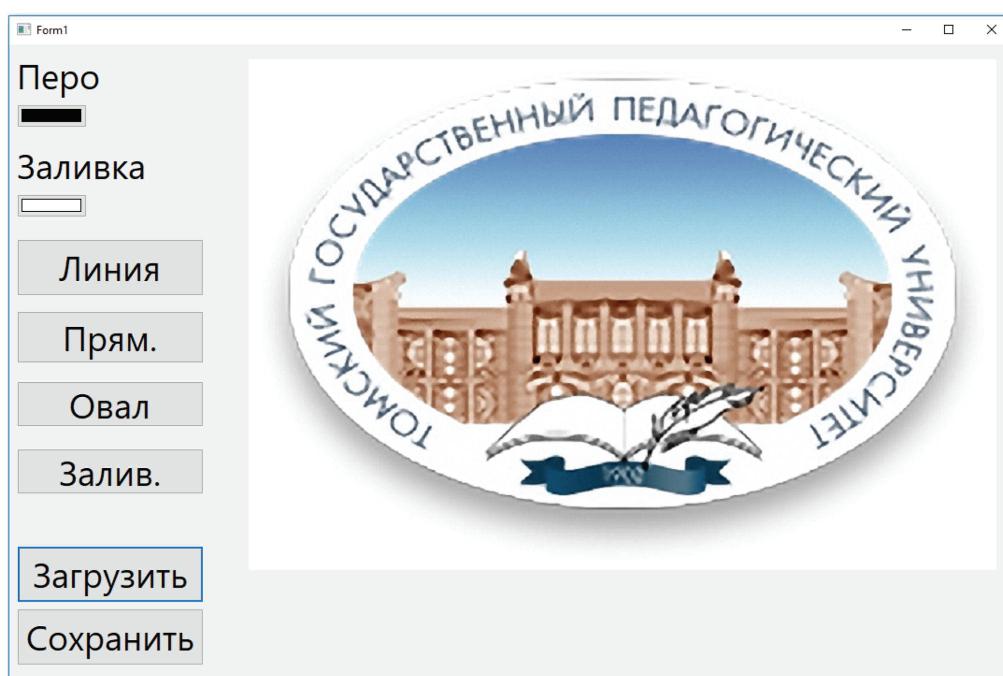
Эллипс – ***Ellipse*** также рисуется по двум точкам: сначала по ним рисуется, невидимый прямоугольник, со сторонами параллельными осям координат (см. выше), а потом в этот прямоугольник вписывается эллипс.

Подсказка: нужно заменить в коде Line на Rectangle или на Ellipse соответственно.

Сделаем кнопку «Загрузить», для загрузки изображения:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
    with TOpenDialog.Create(self) do
    try
        Caption := 'Open Image';
        Options := [ofPathMustExist, ofFileMustExist];
        if Execute then
        Image1.Picture.LoadFromFile(FileName);
    finally
        Free;
    end;
end;
```

Обратите внимание, что **Image1** «растянулся» под размеры изображения.



За загрузку изображения отвечает строка **Image1.Picture.LoadFromFile (FileName);** остальной код служит для построения диалога выбора изображения.

Теперь сделаем кнопку «сохранить»

```
procedure TForm1.Button6Click(Sender: TObject);
begin
   with TSaveDialog.Create(self) do
   try
      Caption := 'Save Image';
      DefaultExt := 'jpg';
      Options := [ofPathMustExist];
      if Execute then
      Image1.Picture.SaveToFile(FileName);
   finally
      Free;
   end;
end;
```

Добавим еще один инструмент – кисть, для этого добавьте еще один **Button** и поле **edit**, которое будет отвечать за толщину линии.

Полю **edit** присвойте по умолчанию значение 1 (линия будет в один пиксель).

На кнопку пропишите выбор кисти (по аналогии с линией), у меня получилось **f:=5.**

Теперь, как будем рисовать «кистью»:

1. Проверяем, выбрана ли кисть
2. Если ЛКМ зажата и движется, то при каждом смещении кисти рисуем круг, радиуса, указанного в поле **edit**
3. Продолжаем до тех пор, пока ЛКМ не будет отпущена

Заведем переменную **Down:boolean,** которая будет следить зажата ЛКМ или отпущена.

В события **FormCreate** и **Image1MouseUp** добавьте строку:

Down:=false;

В событие **MouseDown** строчку:

Down:=true;

Добавим событие движение мыши:

```
procedure TForm1.Image1MouseMove(Sender: TObject;
Shift: TShiftState; X, Y: Integer);
begin
   if (f=5) and (Down) then begin {выбрана кисть и ЛКМ зажата}
   Image1.Canvas.Pen.Color:=ColorButton1.ButtonColor;
   Image1.Canvas.Brush.Color:=ColorButton1.ButtonColor;
```

{в обоих случаях указываем ColorButton1, т.к. рисуем сплошную линию, т.е. контур и заливка должны быть одного цвета}

```
    Image1.Canvas.EllipseC(x,y,strtoint(edit1.text),strtoint(edit1.text));
{рисуем эллипс с центром в точке (x, y) и радиусом сначала указывается радиус по X, потом по Y (для круга они совпадают)}
    end;
end;
```

Задача № 11.1 «Графический редактор. Заливка»

Для реализации заливки воспользуемся алгоритмом поиска в лабиринте, только в нашем случае лабиринт не имеет выхода, поэтому мы обойдем его весь и вернёмся в точку, с которой начали.

1. При выбранной заливке щелкаем ЛКМ в область, которую будем заливать новым цветом.
2. Запоминаем цвет выбранного пикселя – «Цвет 1»
3. Вызываем процедуру «zaliv» для выбранного пикселя
   a. Красим выбранный пиксель в новый цвет «Цвет 2»
   b. Если пиксель слева имеет «Цвет 1», то вызываем процедуру «zaliv» для этого пикселя
   c. Если пиксель сверху имеет «Цвет 1», то вызываем процедуру «zaliv» для этого пикселя
   d. Если пиксель справа имеет «Цвет 1», то вызываем процедуру «zaliv» для этого пикселя
   e. Если пиксель снизу имеет «Цвет 1», то вызываем процедуру «zaliv» для этого пикселя
   f. Если из данного пикселя мы никуда «пойти» не можем, возвращаемся в прошлую процедуру, в то место из которого был вызван новый виток поиска.

Рассмотренная выше процедура называется **рекурсивной**, что вы знаете о таких процедурах?

На кнопку «Заливка» поставьте выбор заливки, в моём случае **f:=4**.

```
procedure TForm1.Image1Click(Sender: TObject);
begin
    if f=4 then begin
    c:=Image1.Canvas.Pixels[MouseX,MouseY];
    zaliv(MouseX,MouseY);
    end;
end;
```

У нас появилось 3 новых переменных:

- с – «Цвет 1», т.е. цвет пикселя в который мы щелкнули мышкой
- MouseX, MouseY – координаты этого пикселя, т.к. событие **Image1Click** не передаёт координаты места, в котором оно произошло, в отличие, например, от **Image1MouseDown** или **Image1MouseUp.**

Поэтому к событию **Image1MouseMove** добавим две новые строки:

```
MouseX:=X;
MouseY:=Y;
```

Теперь создадим процедуру **«zaliv»**, для этого в любом месте добавьте строки (только не внутри другой процедуры!):

```
procedure TForm1.zaliv(X,Y:integer);
begin
    Image1.Canvas.Pixels[x,y]:=ColorButton2.ButtonColor;
    {красим пиксель в новый цвет}
    cl:=Image1.Canvas.Pixels[x-1,y];
    if cl=c then zaliv(x-1,y);
    {запоминаем свет левого пикселя – cl и если он совпал с «цветом
    1» – c, то вызываем для него процедуру «zaliv»}
end;
```

Раз мы создали процедуру её необходимо описать, поэтому в разделе описания процедур, добавьте строчку (после **TForm1 = class(TForm)** и до служебного слова **private**!)

```
procedure zaliv(X,Y:integer);
```

Выполните программу и попробуйте залить что-то, у вас получится луч уходящий налево.

Добавьте еще 3 варианта, для пикселя сверху (*cv*), справа (*cp*) и снизу (*cn*), порядок не имеет значения.

Внимание! Данный метод заливки имеет ряд недостатков.

1. Если цвет заливки совпадет с цветом фона, то процедура уйдет в бесконечный цикл и программа выдаст ошибку
2. Если стоит большое расширение экрана и вы попытаетесь залить большую область, программа также может выдать ошибку
3. Заливка может выйти за пределы видимой части объекта **Image1**, если там что-то есть и вызвать ошибку 2.

Первую ошибку можно исправить, заменив строчку:

```
zaliv(MouseX,MouseY);
```

На условие проверки, совпадает новый цвет с цветом области, если они совпали, то ничего не делаем, т.к. фактически область уже залита этим цветом.

if c<>ColorButton2.ButtonColor then
zaliv(MouseX,MouseY);

Третья ошибка в нашем случае может возникнуть, только если мы искусственно расширим область **Image1**, т.к. в нашем случае всё, что мы не видим залито «невидимым» цветом. Но мы можем залить область большую, чем размеры **Image1**, например, в событии **FormCreate**, указать размеры **Rectangle**, намного больше, чем размеры **Image1** или иным образом.

Вторая ошибка более коварная и единственный вариант – искусственно ограничить количество вызовов процедуры **«zaliv»**.

Задача № 11.2 «Графический редактор. PaintBox»

Теперь давайте визуализируем процесс рисования, чтобы мы видели, как будет выглядеть наш финальный объект. Для этого добавим поверх объекта **Image** новый объект **PaintBox (вкладка Additional)**, с прозрачной заливкой, на котором будем рисовать все промежуточные этапы.

Главная проблема в том, чтобы объект **PaintBox** идеально совпал с объектом **Image**. Добавьте в любое место формы объект **PaintBox1**, после чего нужно его совместить с **Image1**, для этого в событие **FormCreate** добавьте новые строки:

Paintbox1.Left:=Image1.Left;
Paintbox1.Top:=Image1.Top;
Paintbox1.Width:=Image1.Width;
Paintbox1.Height:=Image1.Height;

Внимание! Поскольку объект **Image1** теперь закрыт **PaintBox1** то все события с ним связанные: **Image1MouseDown, Image1MouseMove, Image1MouseUp** и **Image1Click** больше **НЕ** работают, т.к. они не могут теперь произойти! Нужно переписать все эти события под объект **PaintBox1** **(Внимание!!! Не изменить Image1 на PaintBox1, а именно создать эти события!!!)**

Соответственно теперь всё что связано с координатами будет привязано к **PaintBox1**, а с рисованием по-прежнему будет привязано к **Image1.**

Содержимое **Image1MouseDown, Image1MouseMove, Image1MouseUp** и **Image1Click** переносим без изменений в **PaintBox1MouseDown, PaintBox1 MouseMove, PaintBox1MouseUp** и **PaintBox1Click.**

Проверьте, что всё работает.

Теперь перейдём к рисованию промежуточных этапов, алгоритм рисования линии теперь будет выглядеть так:

1. Зажимаем ЛКМ и запоминаем координаты точки – начала линии
2. При движении мыши по экрану на объекте **PaintBox1** рисуется линия из точки, которую мы запомнили в пункте 1, до точки, куда мы переместили мышку. (Обратите внимание, поскольку линия рисуется на прозрачном поле объекта **PaintBox1**, то визуально выглядит, что она рисуется на нашем рисунке **Image1**, но при этом мы наш рисунок не портим)
3. Если мы переместим мышку еще раз, то предыдущая линия должно стереться, а нарисоваться новая линия, в новую точку
4. Когда мы отпустим ЛКМ поле объекта **PaintBox1** должно очиститься, а линия перенестись на наш рисунок – объект **Image1**

Пункты 1 и 4 уже реализованы – линия рисуется, осталось реализовать пункты 2 и 3.

На событие **PaintBox1MouseDown** добавьте 2 строки, для выбора цвета фигуры:

```
PaintBox1.Canvas.Pen.Color:=ColorButton1.ButtonColor;
PaintBox1.Canvas.Brush.Color:=ColorButton2.ButtonColor;
```

Создайте событие

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
    if (f=1) and (Down) then
    PaintBox1.Canvas.Line(DownX, DownY,MouseX,MouseY);
end;
```

Теперь осталось вызвать данное событие, для этого в событие **PaintBox1 MouseMove** добавьте строчку:

```
if Down then PainTBox1.repaint;
```

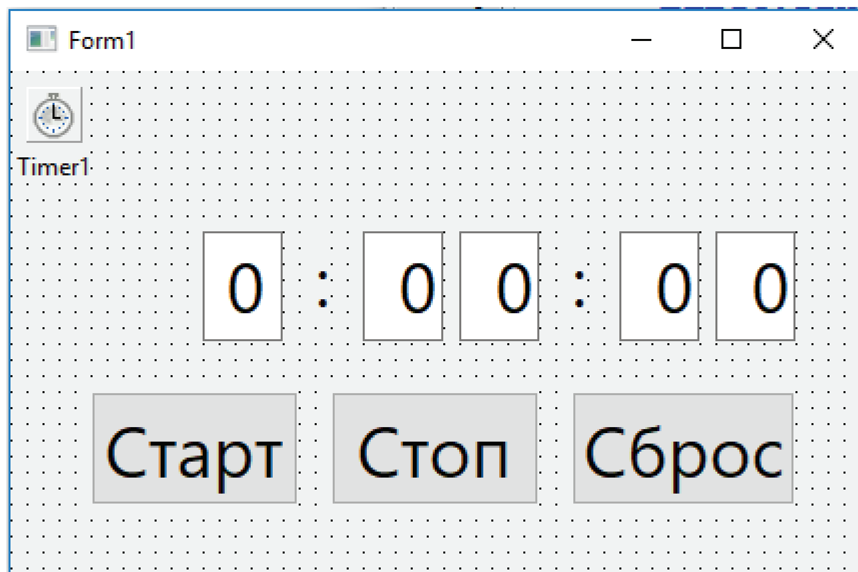
Т.е. если ЛКМ зажата (**Down**) и мышь двигается (происходит событие **PaintBox1MouseMove**), то вызывается событие перерисовывания поля **PainTBox1**, при этом поле очищается, что нам и нужно.

Сделайте аналогично для остальных кнопок!

Задача № 12 «Таймер»

Создаём новый проект. Нам потребуются следующие объекты:

- Timer – 1 шт.
- Button – 3 шт.
- Edit – 5 шт.



Создадим программу – таймер, которая считает количество секунд, минут и часов со старта до остановки. Для этого нам потребуется новый компонент **Timer**, который позволяет вводить необходимые задержки между выполнением тех или иных действий.

У **Timer** всего 4 свойства, нас будут интересовать всего 2:

– Enabled – запущен таймер (true) или остановлен (false)
– Interval – интервал срабатывания таймера в миллисекундах

Поставьте у **Timer** свойство **Enabled** – **true,** а **Interval** – 1000 (1000 миллисекунд = 1 секунде).

У всех **Edit** свойство **Alignment – taRightJustify** (за что отвечает это свойство?), **Text – 0, Enabled – False** (а, за что отвечает это свойство?).

Внимание! Чтобы изменить свойства у нескольких компонент, можно их выбрать с зажатой клавишей «Shift», например, выбрать все **Edit,** и изменить общие, для выбранных компонент свойства. (Можно выбирать не только однотипные компоненты, например, **Edit** и **Button**).

На кнопку «Старт» пропишите следующее событие:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Timer1.Enabled:=true;
end;
```

Т.е. запускаем таймер, после чего, через выбранный интервал (1000 миллисекунд) будет происходить событие **Timer1Timer** (подробнее о нём посмотрим позже).

На кнопку «Стоп» пропишите следующее событие:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
   Timer1.Enabled:=false;
end;
```

Наоборот – останавливаем таймер.

Теперь напишем новое событие

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    Edit1.Text:=IntToStr(StrToInt(Edit1.Text)+1);
end;
```

Где **Edit1** – крайнее справа, оно будет считать единицы секунд (у вас номер **Edit** может отличаться).

Запустите программу и посмотрите, что происходит.

Теперь посмотрим алгоритм работы таймера:

1. Через установленный интервал увеличиваем количество секунд на 1, т.е. увеличиваем число в **Edit1.Text** на единицу.
2. Когда количество секунд станет равно 10, должно произойти 2 события:
   a. Количество десятков секунд увеличиваем на 1
   b. Количество единиц секунд становится равно 0 (на таймере должно получиться 0:00:10).
3. Аналогично, с остальными **Edit**: единицы и десятки минут, часы.

Добавьте на событие **Timer1Timer** строки:

```
if Edit1.Text='10' then begin
    Edit1.Text:='0';
    Edit2.Text:=IntToStr(StrToInt(Edit2.Text)+1);
end;
```

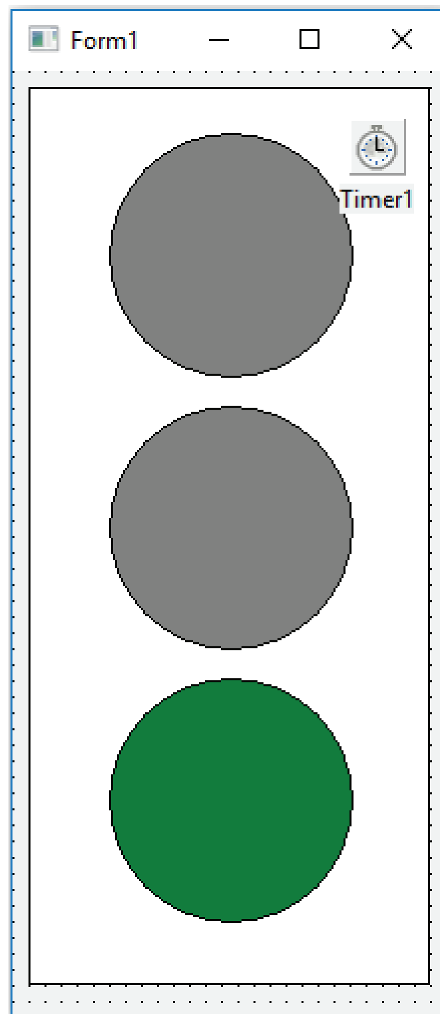
Напишите аналогично для остальных компонент Edit (для ускорения проверки работы таймера измените интервал с 1000 на 10, т.е. время у нас будет бежать в 100 раз быстрее).

Самостоятельно напишите код на кнопку «Сброс».

Задача № 13 «Светофор»

Создаём новый проект. Нам потребуются следующие объекты:

– Timer – 1 шт.
– Shape – 4 шт.



Shape (вкладка **Additional**) – это фигура, которая может принимать форму: круга, треугольника, прямоугольника и т.д.

Для **Shape1** поставьте свойство **Shape – stRectangle**, это будет наш светофор.

Добавим сигналы светофора, для этого у **Shape2, Shape3, Shape4** поставьте свойство **Shape – stCircle**, цвет у двух верхних поставим **clGray** (серый), а у нижнего **clGreen** (зелёный). (Нужно точное название цвета или его код!).

Чтобы **Shape** не перекрывали друг друга, выберите «светофор» (**Shape1**), ПКМ далее Z-порядок и выберите «Поместить сзади»

Теперь определимся со временем «горения» сигналов светофора, например, зелёный и красный у нас будут гореть 3 секунды, а жёлтый всего одну. Но у таймера есть лишь одно значение интервала, поэтому нужно завести переменную, которая будет «накапливать» секунды.

На событие **FormCreate** добавьте строку:

S:=0;

Свойство **Timer1 – Interval** установите в 1 секунду.

Теперь разберёмся с алгоритмом:

1. Если горит зелёный сигнал и прошло 3 секунды (S=3), то:
   a. красим зелёный сигнал серым цветом
   b. Обнуляем, счетчик времени (S:=0)
   c. Зажигаем жёлтый сигнал
2. Если горит жёлтый сигнал и прошло 1 секунда (S=1), то:
   a. красим жёлтый сигнал серым цветом
   b. Обнуляем, счетчик времени (S:=0)
   c. Зажигаем красный сигнал
3. Если горит красный сигнал и прошло 3 секунды (S=3), то:
   a. красим красный сигнал серым цветом
   b. Обнуляем, счетчик времени (S:=0)
   c. Зажигаем зелёный сигнал

Для зелёного сигнала код будет выглядеть так:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    S:=S+1; {увеличиваем счетчик времени на 1 секунду}
    if (s=3) and (shape4.Brush.Color=clGreen) then begin
        shape3.Brush.Color:=clYellow;
        shape4.Brush.Color:=clGray;
        s:=0;
    end;
end;
```

Напишите, аналогично для жёлтого и красного сигналов.

Единственное, в нашем случае, после красного сигнала сразу включается зелёный сигнал, пропуская жёлтый. Это можно исправить, введя еще одну переменную, которая будет отслеживать прошлое состояние светофора (например, горел до жёлтого сигнала зелёный сигнал или нет).

Алгоритм для жёлтого сигнала изменится следующим образом:

1. Если (горит жёлтый сигнал) и (прошло 1 секунда (S=1)) и (предыдущий сигнал был зелёный), то:
   a. красим жёлтый сигнал серым цветом
   b. обнуляем, счетчик времени (S:=0)
   c. зажигаем красный сигнал
2. Если (горит жёлтый сигнал) и (прошло 1 секунда (S=1)) и (предыдущий сигнал **НЕ** был зелёный), то:
   a. красим жёлтый сигнал серым цветом
   b. обнуляем, счетчик времени (S:=0)
   c. зажигаем зелёный сигнал

Для этого заводим новую переменную и на зелёный сигнал добавляем строку:

```
Z:=true;
```

Жёлтый сигнал заменяем на:

```
if (s=1)and(shape3.Brush.Color=clYellow)and(z)then begin
    shape2.Brush.Color:=clRed;
    shape3.Brush.Color:=clGray;
    s:=0;
    z:=false;
end;
```

И добавьте еще один вариант для жёлтого сигнала, на случай **z=false,** т.е. зажигаем уже не красный, а зелёный.

```
if (s=1)and(shape3.Brush.Color=clYellow)and(z=false)then begin
    shape4.Brush.Color:=clGreen;
    shape3.Brush.Color:=clGray;
    s:=0;
end;
```

У красного сигнала получится так:

```
if (s=3)and(shape2.Brush.Color
    =clRed)then begin
    shape2.Brush.Color:=clGray;
    shape3.Brush.Color:=clYellow;
    s:=0;
end;
```

Рекомендуемая литература

1. Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие для вузов / П. Б. Хорев. – Москва : Академия, 2011. – 446 с.
2. Архангельский, А. Я. Язык Pascal и основы программирования в Delphi : учебное пособие для вузов / А. Я. Архангельский. – Москва : Бином, 2004. – 495 с.
3. Культин, Н. Б. Delphi.NET в задачах и примерах : сборник программ и задач / Н. Б. Культин. – Санкт-Петербург : БХВ-Петербург, 2006. – 255 с.
4. Фаронов, В. В. Delphi. Программирование на языке высокого уровня : учебник для вузов / В. В. Фаронов. – Санкт-Петербург : Питер, 2009. – 639 с.
5. Васильев, А. Н. Java. Объектно-ориентированное программирование для магистров и бакалавров : базовый курс по объектно-ориентированному программированию / А. Н. Васильев. – Санкт-Петербург : Питер, 2012. – 395 с.

Учебное издание

Карташов Денис Васильевич
Непомнящая Людмила Александровна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие

Ответственный за выпуск: Л. В. Домбраускайте
Технический редактор: Н. Н. Сафронова

Бумага: офсетная
Печать: трафаретная
Усл. печ. л.: 1,7
Уч. изд. л.: 3,1

Сдано в печать: 10.09.2019 г.
Формат: 64×80/16
Заказ: 1436/У
Тираж: 100 экз.

Издательство Томского государственного педагогического университета
634061, г. Томск, ул. Киевская, 60
Отпечатано в типографии Издательства ТГПУ
г. Томск, ул. Герцена, 49. Тел. (3822) 31-14-84.
e-mail: tipograf@tspu.edu.ru