

А. И. Новиков

**АЛГОРИТМИЗАЦИЯ
И ПРОГРАММИРОВАНИЕ**

**ВЫПОЛНЕНИЕ КОНТРОЛЬНЫХ РАБОТ
ДЛЯ СТУДЕНТОВ ПЕРВОГО КУРСА
ЗАОЧНОЙ ФОРМЫ ОБУЧЕНИЯ**

Учебно-методическое пособие

**Санкт-Петербург
2024**

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«Санкт-Петербургский государственный университет
промышленных технологий и дизайна»
Высшая школа технологии и энергетики**

А. И. Новиков

**АЛГОРИТМИЗАЦИЯ
И ПРОГРАММИРОВАНИЕ**

**ВЫПОЛНЕНИЕ КОНТРОЛЬНЫХ РАБОТ
ДЛЯ СТУДЕНТОВ ПЕРВОГО КУРСА
ЗАОЧНОЙ ФОРМЫ ОБУЧЕНИЯ**

Учебно-методическое пособие

Утверждено Редакционно-издательским советом ВШТЭ СПбГУПТД

Санкт-Петербург
2024

УДК 004.432
ББК 32.973
Н731

Рецензенты:

кандидат технических наук, доцент, заведующий кафедрой прикладной математики и информатики Высшей школы технологии и энергетики Санкт-Петербургского государственного университета промышленных технологий и дизайна

И. В. Ремизова;

доктор технических наук, профессор, заведующий кафедрой автоматизации процессов химической промышленности Санкт-Петербургского государственного технологического института (Технического университета)

Л. А. Русинов

Новиков, А. И.

Н731 Алгоритмизация и программирование. Выполнение контрольных работ для студентов первого курса заочной формы обучения: учебно-методическое пособие / А. И. Новиков. — СПб.: ВШТЭ СПбГУПТД, 2024. — 118 с.

Учебно-методическое пособие соответствует программам и учебным планам дисциплины «Алгоритмизация и программирование» для студентов заочной формы обучения, обучающихся по направлению подготовки: 09.03.03 «Прикладная информатика», а также может быть полезно при обучении по направлениям подготовки: 01.03.02 «Прикладная математика и информатика», 15.03.04 «Автоматизация технологических процессов и производств» и 27.03.04 «Управление в технических системах».

В учебно-методическом пособии изложены основы работы со средой разработки Delphi и синтаксис языка Паскаль, приведены примеры кода для работы с событийно-ориентированным программированием. Учебно-методическое пособие содержит разделы для самостоятельного углубленного изучения.

Учебно-методическое пособие предназначено для подготовки бакалавров заочной формы обучения. Отдельные разделы пособия могут быть полезны магистрантам и аспирантам.

УДК 004.432
ББК 32.973

© Новиков А. И., 2024
© ВШТЭ СПбГУПТД, 2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	4
1. ОСНОВЫ PASCAL	5
1.1. Операции и операторы в Pascal.....	7
1.2. Комментарии.....	12
1.3. *Особенности установки Delphi 7	12
1.4. Заглядывая в Delphi.....	13
1.5. Запуск программы и возможные ошибки	21
1.6. Объявление переменных и констант	22
1.7. Преобразование типов	26
1.8. Математические функции	28
1.9. РАБОТА № 1 «Разработка программы «Калькулятор»	30
1.10. *РАБОТА № 1 (дополнение) «Калькулятор с двумя полями»	32
1.11. Строковые функции	33
1.12. Создание собственных процедур и функций	35
1.13. Модули Pascal	37
1.14. Некоторые полезные процедуры	40
1.15. Условный оператор if.....	41
1.16. Цикл for	43
2. ОСНОВЫ DELPHI	45
2.1. Расширения файлов Delphi.....	46
2.2. *Версии Delphi.....	47
2.3. Библиотека VCL	48
2.4. Свойства визуальных компонентов.....	51
2.5. Изменение иконки приложения	59
2.6. Невизуальные компоненты Delphi	59
2.7. Диалоговые окна.....	61
2.8. Механизм Actions	65
2.9. РАБОТА № 2 «Разработка текстового редактора».....	69
2.10. *Action-ы с привязками	74
2.11. *ActionManager	76
2.12. Событийно-ориентированное программирование.....	82
2.13. События визуальных компонентов	85
2.14. События невидимых компонентов	92
2.15. События формы	95
2.16. Обзор визуальных компонентов Delphi	97
2.17. Графический интерфейс пользователя.....	108
2.18. Свойства и методы компонентов.....	109
2.19. Свойства и методы полей Edit, Memo и RichEdit	110
2.20. Диалоговые окна Application.MessageBox.....	114

ВВЕДЕНИЕ

Паскаль (Pascal) – один из наиболее известных языков программирования, используется для обучения программированию в старших классах и на первых курсах вузов, является основой для ряда других языков. Наряду с такими языками как, например, С, С++, Java, Python, PHP, относится к языкам программирования 3-го поколения (3GL). К языкам 2-го поколения (2GL) относятся Ассемблеры, т.е. языки низкого уровня.

Отличие Паскале-подобных языков от большинства других в том, что Паскаль называет вещи своими именами. Так если создается функция, то первым словом будет `function`; для процедуры – `procedure`; для переменной – `var`; для константы – `const`; для типа – `type`; для модуля первым словом будет `unit`, для библиотеки – `library` и т.д. Что (с точностью до перевода) дает четкое понимание того, что создается.

Delphi – императивный, структурированный, объектно-ориентированный, компилируемый высокоуровневый язык программирования общего назначения со строгой статической типизацией переменных. Является диалектом языка Паскаль (его объектно-ориентированной версии, т.е. Object Pascal). Основная область использования – написание прикладного программного обеспечения.

VCL (Visual Component Library – Библиотека визуальных компонентов) – объектно-ориентированная библиотека для разработки программного обеспечения, созданная компанией Borland. Входит в комплект поставки Delphi, C++ Builder и RAD Studio, и является частью среды разработки.

Событийно-ориентированное программирование – парадигма программирования, в которой выполнение программы определяется событиями (Events) – т.е. действиями пользователя (клавиатура, мышь, сенсорный экран), а также сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета). Событийно-ориентированное программирование применяется при построении пользовательских интерфейсов.

** Пособие содержит разделы для самостоятельного углубленного изучения, их названия отмечены звездочкой.*

1. ОСНОВЫ PASCAL

Delphi (Делфи) – императивный, структурированный, объектно-ориентированный, компилируемый высокоуровневый язык программирования общего назначения со строгой статической типизацией переменных. Является диалектом языка **Паскаль** (его объектно-ориентированной версии, т.е. **Object Pascal**). Основная область использования – написание прикладного программного обеспечения.

!!! Говоря: «Написан на Delphi» и «Написан на Паскале» – мы будем иметь в виду одно и то же (если явно не указано обратное).

Паскаль (Pascal) – один из наиболее известных языков программирования, используется для обучения программированию в старших классах и на первых курсах вузов, является основой для ряда других языков. Наряду с такими языками как, например, **C, C++, Java, Python, PHP**, относится к языкам программирования 3-го поколения (3GL). *К языкам 2-го поколения (2GL) относятся Ассемблеры, т.е. языки низкого уровня.*

Паскаль создан Никлаусом Виртом в **1970** году. По мнению Вирта, язык должен способствовать дисциплинированному программированию, поэтому, наряду со строгой типизацией, в Паскале сведены к минимуму возможные синтаксические неоднозначности, а сам синтаксис автор постарался сделать интуитивно понятным (даже при первом знакомстве с языком). При создании языка (в отличие от языка Си) не ставилось задачи обеспечить максимальную производительность или краткость исходного кода. Изначально язык ставил во главу угла высокую читаемость исходного кода, т.к. был предназначен для обучения программированию.

Отличие Паскале-подобных языков от большинства других в том, что **Паскаль называет вещи своими именами**. Так, если создается функция, то первым словом будет **function**; для процедуры – **procedure**; для переменной – **var**; для константы – **const**; для типа – **type**; для модуля первым словом будет **unit**, для библиотеки – **library** и т.д. Что (с точностью до перевода) дает четкое понимание того, что создается.

Все языки делятся на две группы: «общего назначения» и «предметно-ориентированные».

Язык программирования общего назначения (general-purpose programming language) – это язык программирования для создания программного обеспечения в самых разных областях применения.

Например: **C, C++, Delphi/Pascal, Java, PHP, Python, Visual Basic** и др.

Предметно-ориентированный язык (domain-specific language – «язык, специфический для предметной области»), «**язык специального назначения**» – это компьютерный язык, специализированный для конкретной области применения (в противоположность языку «**общего назначения**», применимому к широкому спектру областей и не учитывающему особенности конкретных сфер знаний). Построение такого языка и его структура данных отражают специфику решаемых с его помощью задач.

Например:

- **HTML** – для разметки документов (Интернет-страниц);
- **SQL** – для СУБД (Систем Управления Базами Данных);
- **TeX/LaTeX** – для компьютерной верстки текстовых документов;
- **Verilog** и **VHDL** – для описания аппаратного обеспечения;
- **G-код** – язык программирования станков с числовым программным управлением (ЧПУ);
- и т.д.

Деление языков программирования на языки общего назначения и предметно-ориентированные весьма условно.

Языки бывают компилируемые и интерпретируемые.

Компилируемый язык программирования – язык программирования, исходный код которого преобразуется компилятором в машинный код и записывается в файл (бинарный файл) с особым расширением или заголовком для последующей идентификации этого файла, как исполняемого операционной системой (для Windows это файлы с расширением ***.exe**).

К преимуществам компилируемых языков относится:

- высокая скорость работы программы (в сотни раз быстрее интерпретируемых языков);
- меньший объем требуемой оперативной памяти и меньший размер исполняемого файла на диске. *В этом сравнении необходимо считать суммарный размер всех файлов, необходимых для запуска программы, а не только кода программы пользователя, или полученного бинарного файла;*
- возможность (в большинстве случаев) запуска **exe**-файла без установки дополнительного ПО (среды разработки, интерпретатора, виртуальной машины и т.п.).

Недостатком является то, что бинарные файлы создаются под конкретный процессор и операционную систему.

К компилируемым языкам относятся, например: **Ассемблер, C/C++, Delphi/Pascal/Ada, Fortran, Java**.

Интерпретируемыми языками программирования являются, например: **JavaScript, PHP, Bash, BAT**-файлы Windows, **Python, Visual Basic for Applications (VBA)**.

Как правило, к интерпретируемыми языками относятся языки сценариев (**script**, скрипты, скриптовые языки). Достоинством интерпретируемых языков является их независимость от среды исполнения (кроссплатформенность).

Для некоторых языков существует как компилятор, так и интерпретатор.

Некоторые языки (например, **Java**), компилируются для исполнения не на конкретном процессоре, а на специальной виртуальной машине. Это позволяет получить преимущества обоих способов, высокую скорость и кроссплатформенность (но не размер файлов или свободу от дополнительного ПО, т.к. в этом случае требуется установка и запуск виртуальной машины).

Кроме того, языки делятся на **регистрозависимые** (чувствительные к регистру) и **регистронезависимые** (нечувствительные к регистру).

Нечувствительными к регистру являются: **Pascal, SQL, Fortran, HTML**, большинство **ассемблеров**, имена файлов в операционной системе **Windows**, доменные имена и адреса электронной почты (*например, «Ivanov@Mail.Ru» и «ivanov@mail.ru» считаются одним адресом*).

Чувствительными к регистру являются: **C, C++, Java, C#, PHP, Python, TeX, XML**, ввод паролей, имена файлов в операционной системе **Linux** (*например, здесь в одной папке могут одновременно существовать папки «Новая папка» и «новая папка», т.к. это разные имена*).

Регистрозависимость встречается не только в компьютерной технике, так, например, в системе единиц Си «МПа» означает «мегапаскаль», а «мПа» означает «миллипаскаль».

!!! Несмотря на то, что язык **Паскаль** позволяет использовать одно и то же имя в любом регистре (например, «Caption», «caption» и «CAPTION»), но одновременное использование в студенческих работах различного написания одного и того же имени – будет ошибкой оформления! Более того, те имена, которые уже объявлены разработчиками **Delphi**, нужно писать именно в том виде, как они были объявлены (т.е. из приведенных выше примеров, правильным будет только «Caption»).

1.1. Операции и операторы в Pascal

Большинство операций в языке **Паскаль** имеют следующую запись (рис. 1).



Рисунок 1 – Запись операций

Если операнда два, то оператор называется «**бинарным**», а соответствующая операция «**бинарной**».

Если операнд один, то операция и оператор называются «**унарными**».

Также в программировании встречаются операции с тремя операндами – «**тернарные**» (но в **Delphi** таких нет).

Арифметические операции

Все арифметические операции (табл. 1) могут применяться к целым числам (**Integer**), а некоторые также и к числам с плавающей запятой (**Real**).

Числа с плавающей запятой также могут называться числами с плавающей точкой, действительными числами или вещественным числам. Вещественные числа включают в себя и целые числа.

Таблица 1 – Арифметические операции

Оператор	Операция	Пример	Типы операндов (входные значения)	Тип результата (выходное значение)
+	Сложение	$7 + 4 = 11$	Integer, Real (целые и нецелые)	Integer, Real (целые и нецелые)
-	Вычитание	$7 - 4 = 3$	Integer, Real (целые и нецелые)	Integer, Real (целые и нецелые)
*	Умножение	$7 * 4 = 28$	Integer, Real (целые и нецелые)	Integer, Real (целые и нецелые)
/	Деление	$7 / 4 = 1,75$	Integer, Real (целые и нецелые)	Real (только вещественные)
div	Целочисленное деление	$7 \text{ div } 4 = 1$	Integer (только целые)	Integer (только целые)
mod	Остаток от целочисленного деления	$7 \text{ mod } 4 = 3$	Integer (только целые)	Integer (только целые)

Унарные арифметические операции

Унарные арифметические операции (табл. 2) имеют только один операнд, записываемый после оператора, например, **-4**.

Таблица 2 – Унарные арифметические операции

Оператор	Операция	Пример	Типы операнда (входное значение)	Тип результата (выходное значение)
-	Минус	$-4 - 3 = -7$	Integer, Real (целые и нецелые)	Integer, Real (целые и нецелые)
+	Плюс	$+4 - 3 = 1$	Integer, Real (целые и нецелые)	Integer, Real (целые и нецелые)

Операции сравнения

Операции сравнения (табл. 3), они же Операции отношения, занимают промежуточное положение между Арифметическими и Логическими операциями. На вход им подаются численные значения (либо текстовые строки **String**), а на выходе получается логическое значение **Boolean**. Логическое значение (Булево значение) может принимать только одно из двух значений **True** и **False** (они же «Да» или «Нет», «Истина» или «Ложь», «1» или «0»).

Таблица 3 – Операции сравнения

Оператор	Операция	Пример	Типы операнда (входное значение)	Тип результата (выходное значение)
=	Равно	5 = 7 – Нет	Integer, Real, String (целые и нецелые, а также текстовые строки)	Boolean (логические)
>	Больше	5 > 7 – Нет		
<	Меньше	5 < 7 – Да		
>=	Больше или равно	5 >= 7 – Нет		
<=	Меньше или равно	5 <= 7 – Да		
<>	Не равно	5 <> 7 – Да		

Логические операции

Логические операции (табл. 4), они же Булевы операции, применяются к логическим значениям и возвращают результат также в виде логического значения.

Таблица 4 – Логические операции

Оператор	Операция	Пример	Типы операнда (входное значение)	Тип результата (выходное значение)
not	«Не», Логическое отрицание, Инверсия	not True = False	Boolean (логические)	Boolean (логические)
and	«И», Логическое умножение	True and False = False		
or	«ИЛИ», Логическое сложение	True or False = True		
xor	«Исключающее ИЛИ», «Искл. ИЛИ», Сложение по модулю 2	True xor True = False		

Работа логических операций демонстрируется при помощи «Таблиц истинности». В таблицах истинности входы отделяются от выходов жирной или двойной линией.

Таблица истинности для логического умножения представлена в таблице 5.

Таблица 5 – Логическое умножение

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Таблица истинности для логического сложения представлена в таблице 6.

Таблица 6 – Логическое сложение

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

Таблица истинности для сложения по модулю 2 представлена в табл. 7.

Таблица 7 – Сложение по модулю 2

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

!!! Стоит обратить внимание, что логическое сложение, отличается от сложения по модулю 2, только последней строчкой.

Выше были рассмотрены бинарные логические операции. Но логическое отрицание (табл. 8) отличается от них, т.к. является унарным.

Таблица 8 – Логическое отрицание

A	not A
0	1
1	0

Побитовые операции

Побитовые операции (табл. 9), они же поразрядные операции, аналогичны логическим операциям, но применяются к целым числам. Результат также будет целым числом. Для вычисления побитовых операций необходимо:

- перевести входные значения из десятичной (**DEC**) в двоичную (**BIN**) систему;
- применить требуемую логическую операцию независимо к каждому из битов (разрядов);
- перевести полученный результат из двоичной (**BIN**) системы обратно в десятичную (**DEC**).

Таблица 9 – Побитовые операции

Оператор	Операция	Пример	Типы операнда (входное значение)	Тип результата (выходное значение)
not	Побитовое отрицание «Не»	Для 4-битных беззнаковых чисел: not 5 = 10 (not b0101 = b1010). Для знаковых чисел: not 5 = -6	Integer (целые)	Integer (целые)
and	Побитовое «И»	5 and 3 = 1 (b101 and b011=b001)		
or	Побитовое «ИЛИ»	5 or 3 = 7 (b101 or b011=b111)		
xor	Побитовое «Искл. ИЛИ»	5 xor 3 = 6 (b101 xor b011=b110)		
shl	Поразрядный сдвиг влево (shift left)	12 shl 3 = 96 (b1100 shl 3 = b1100000, или $12 * 2^3 = 96$)		
shr	Поразрядный сдвиг вправо (shift right)	12 shr 2 = 3 (b1100 shr 2 = b11, или $12 \text{ div } 2^2 = 3$)		

Кроме четырех логических, здесь добавляются еще две операции сдвигов:

- сдвиг на каждый разряд влево равносильно умножению на 2;
- сдвиг на каждый разряд вправо равносильно делению на 2 (деление целочисленное, т.е. остаток отбрасывается).

Строковые операции

Для объединения текстовых строк используется операция конкатенации (табл. 10).

Таблица 10 – Строковые операции

Оператор	Операция	Пример	Типы операндов (входные значения)	Тип результата (выходное значение)
+	Конкатенация (<i>сложение строк</i>)	'7' + '4' = '74'	String (строка текста)	String (строка текста)

!!! Строковые значения записываются в одинарных кавычках!

Оператор присваивания

Оператор присваивания (табл. 11) применяется к любым типам данных.

Таблица 11 – Оператор присваивания

Оператор	Операция	Пример
:=	Присваивание значения переменной	x := 5 + 3

Позволяет записать значение (в примере выше это **8**) в некоторую ячейку памяти (в примере выше это переменная с именем **x**).

1.2. Комментарии

В **Delphi** имеется несколько видов комментариев:

- однострочный комментарий. Начинается с двух «двоеточий» и заканчивается в конце строки, например:

//Однострочный комментарий

- многострочный комментарий. Записывается в фигурных скобках, например:

*{Многострочный
комментарий}*

На самом деле этот комментарий не обязан занимать несколько строк, он может записываться и в одну строку, например:

{Комментарий}

Более того, в таком виде этот комментарий может использоваться в начале или в середине строки (*в отличие от однострочного комментария*).

!!! В студенческих работах использование комментариев для пояснения работы программы является обязательным требованием!

1.3. *Особенности установки Delphi 7

По умолчанию **Delphi 7** устанавливается в папки, представленные на рисунке 2.

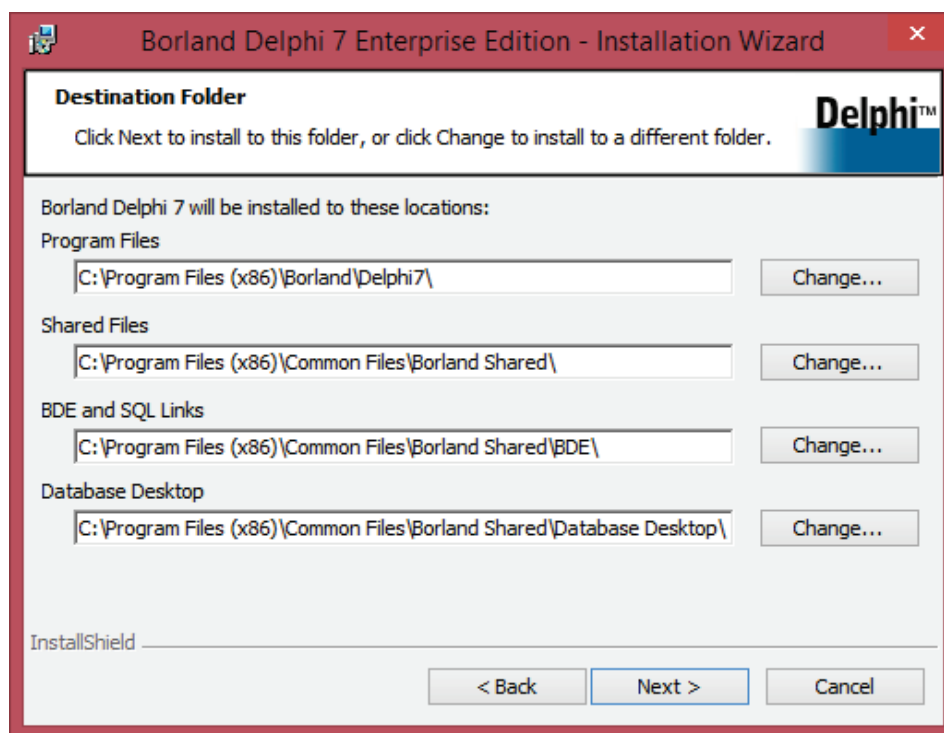


Рисунок 2 – Папки для установки по умолчанию

Но в новых версиях **Windows** не стоит устанавливать старые программы в папку «**Program Files**». В противном случае потребуется запускать **Delphi** от имени Администратора.

Поэтому, изменим папки для установки следующим образом (рис. 3).

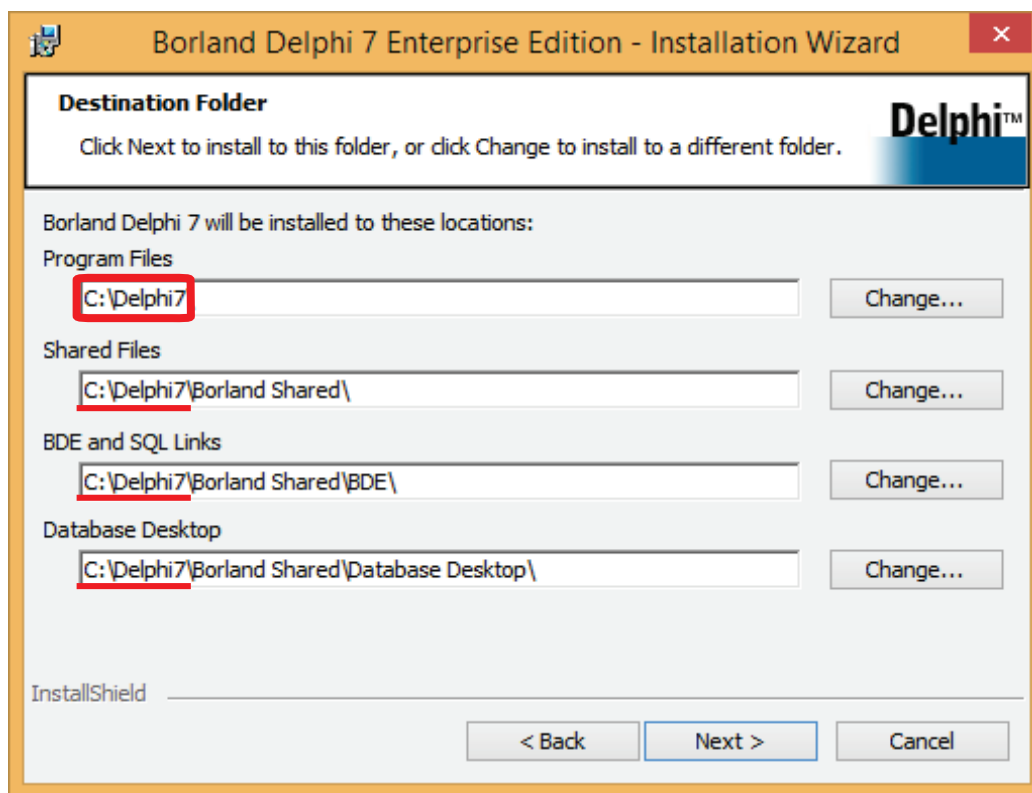


Рисунок 3 – Папки для установки

1.4. Заглядывая в Delphi

Хотя в данной главе изучается **Паскаль**, для тестирования новых операторов и функций нам придется сразу немного заглянуть вперед и ознакомиться со средой разработки **Delphi**. Внешний вид среды будет рассматриваться на примере **Delphi** версии 7.

На данном этапе в среде разработки выделим 3 основных части (рис. 4):

- Форма – т.е. будущее окно нашего приложения;
- Палитра компонентов (сверху), из которой на Форму добавляются новые компоненты;
- Инспектор объектов (слева), в котором производится настройка компонентов.

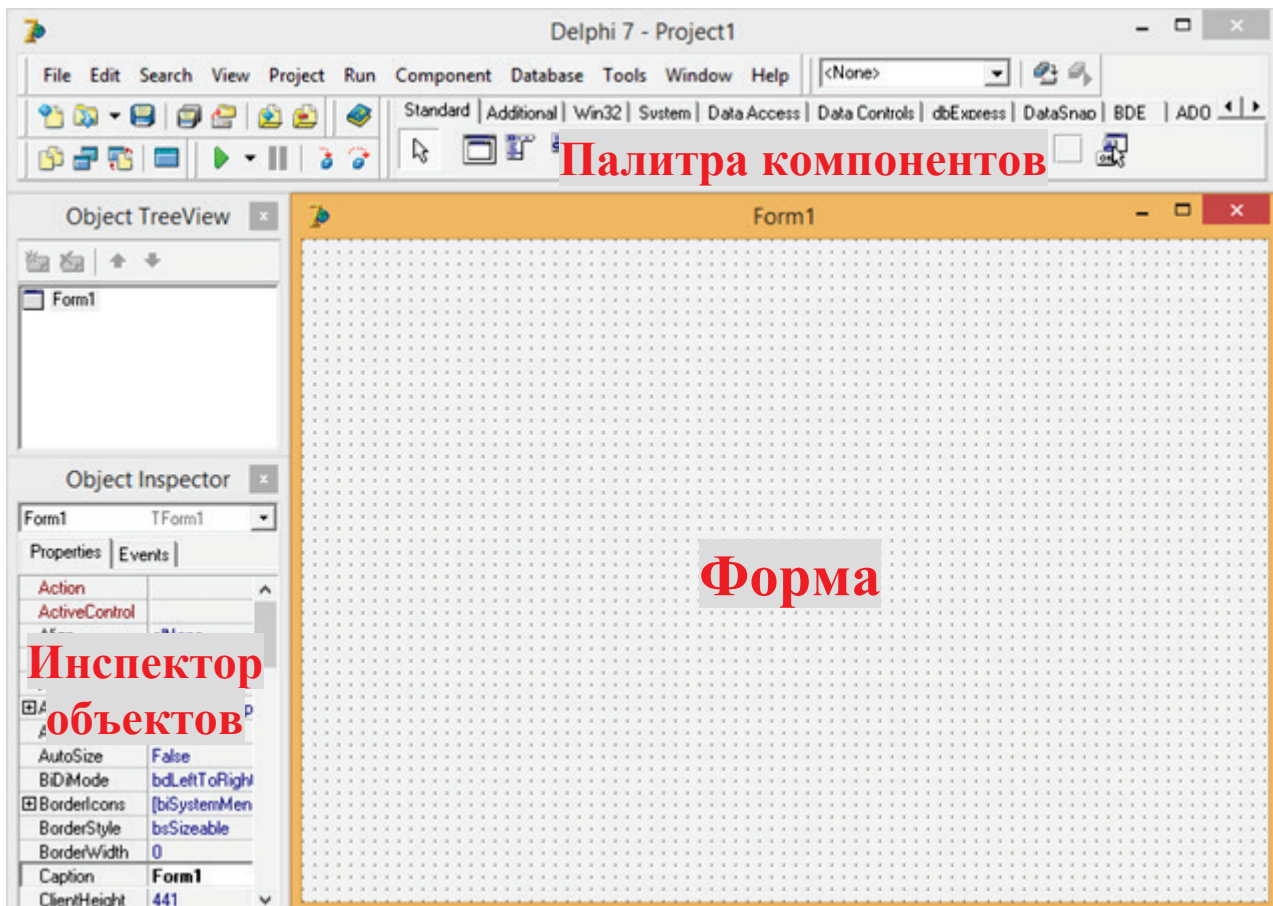


Рисунок 4 – Внешний вид среды Delphi 7

Разработка графического интерфейса

Сразу изменим название окна (Формы), например, на «Калькулятор» (рис. 5). Для этого в Инспекторе объектов введем новое значение для свойства **Caption** (рис. 6).

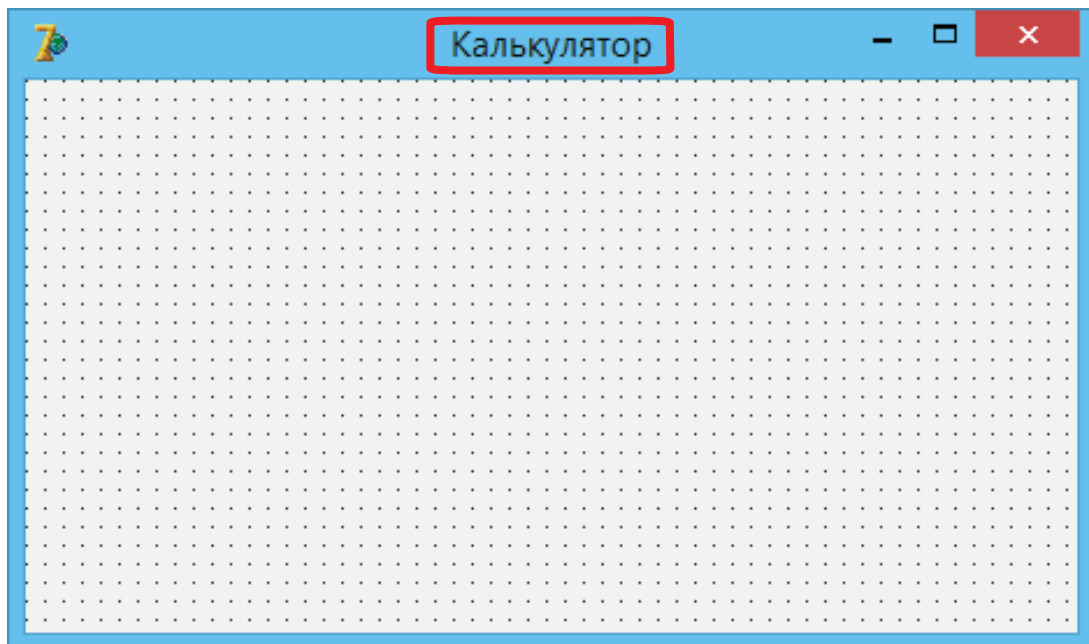


Рисунок 5 – Новое название окна

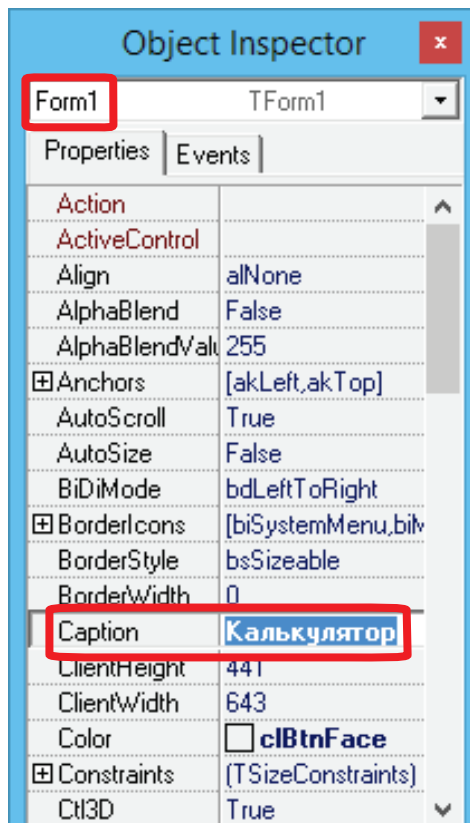


Рисунок 6 – Изменение названия окна

Новые компоненты добавляются на Форму из Палитры компонентов, расположенной сверху (рис. 7).

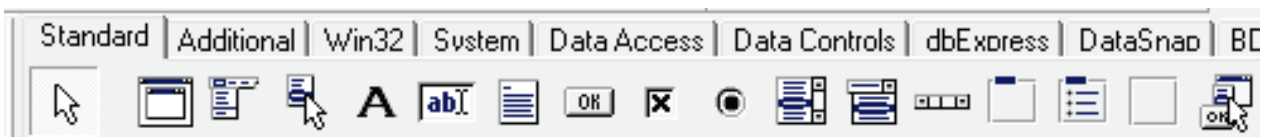




Рисунок 7 – Палитра компонентов Delphi 7

На данном этапе нас интересуют всего два компонента:

-  **Button** – Кнопка;
-  **Edit** – Поле ввода.

!!! В новых версиях Delphi (например, в **Delphi XE8**) Палитра компонентов расположена не сверху, а справа (рис. 8).

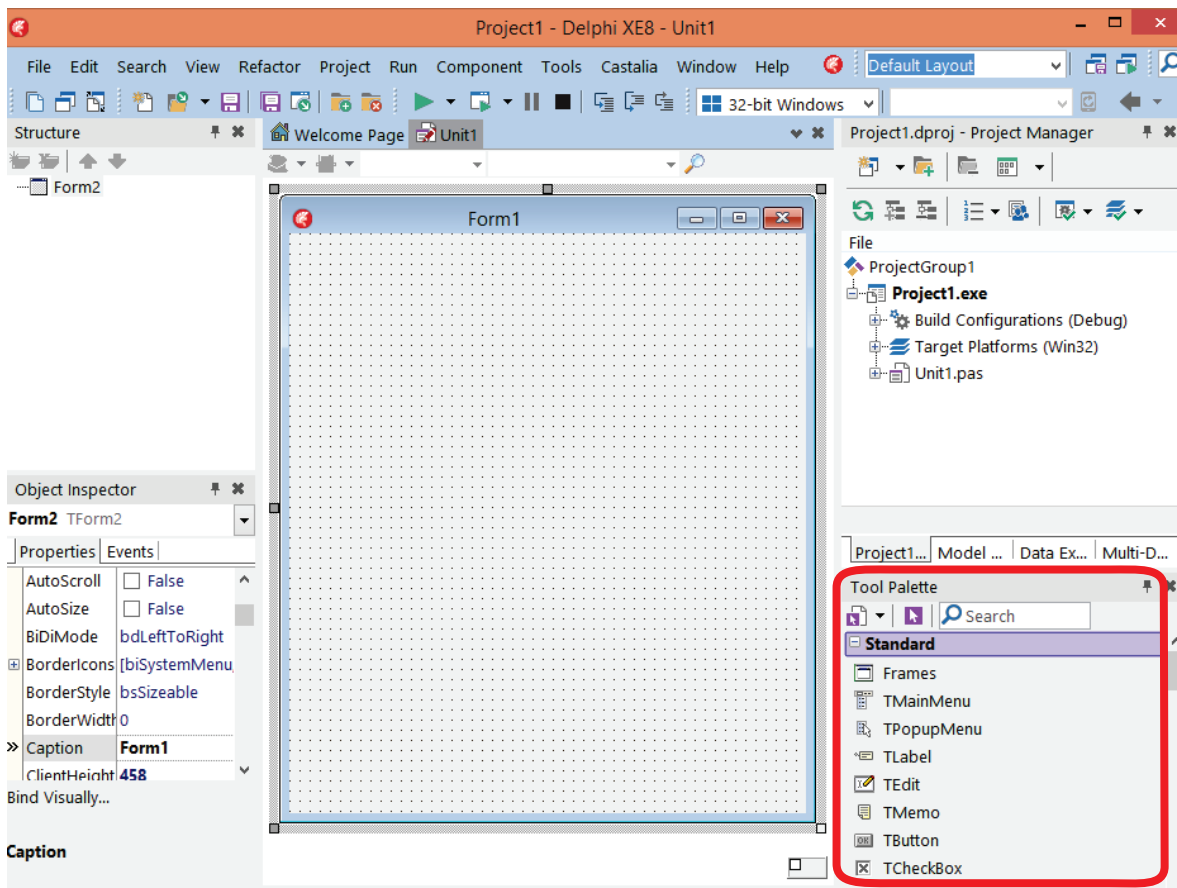


Рисунок 8 – Палитра компонентов в Delphi XE8

Добавим на Форму два Поля ввода и Кнопку (рис. 9).

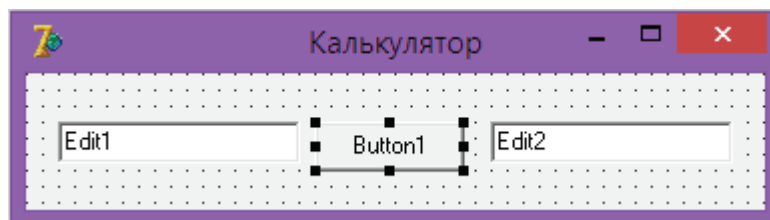


Рисунок 9 – Форма с добавленными компонентами

Перемещать компоненты по Форме и изменять их размеры можно при помощи мышки. Но также можно изменять размеры и положение через Инспектор объектов – свойства **Left**, **Top**, **Width** и **Height** (Слева, Сверху, Ширина и Высота).

Для того чтобы в Инспекторе объектов отобразился нужный компонент, необходимо вначале выделить его на Форме. Изменим надписи (рис. 10) на добавленных компонентах. Для Кнопки это свойство **Caption**, а для Полей ввода это свойство **Text**.

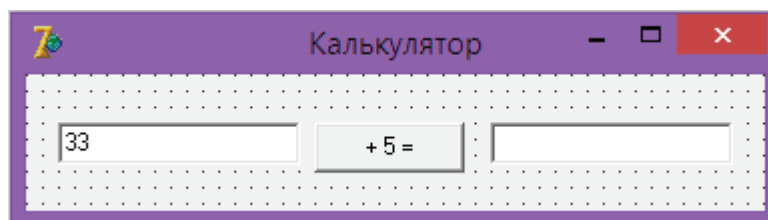


Рисунок 10 – Изменение надписей на компонентах

Через Инспектор объектов можно настроить и другие свойства, например, Шрифт (**Font**). Для того чтобы увеличить размер шрифта (рис. 11) используется свойство **Size** – размер, для цвета – **Color**, свойства **fsBold** и **fsItalic** позволяют сделать текст жирным или курсивным.

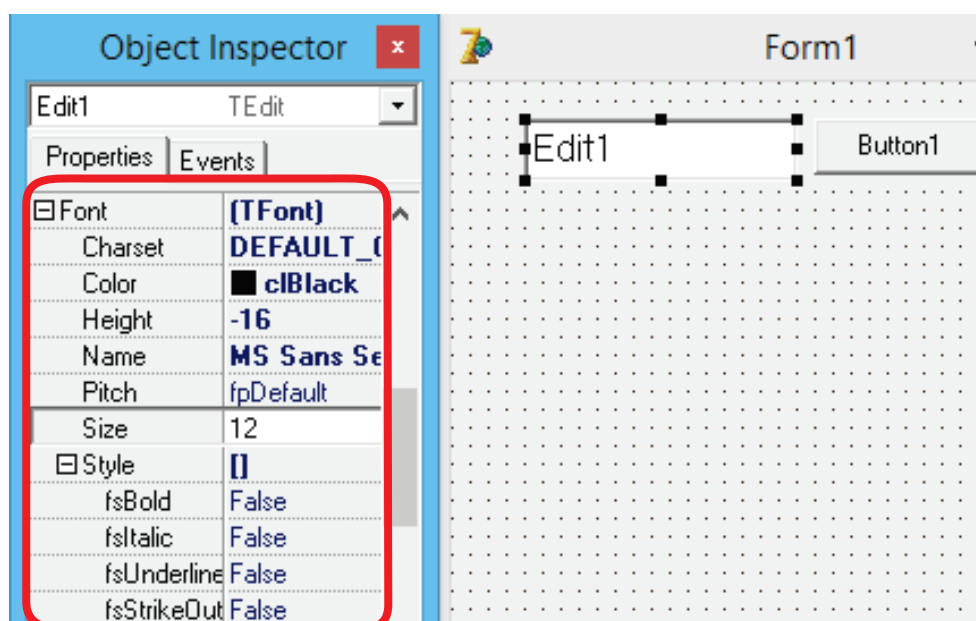


Рисунок 11 – Настройка шрифта

*Кнопка с изображением

При желании, вместо кнопки с текстовой надписью, можно использовать кнопку с изображением. Для этого вместо **Button**, необходимо добавить другой тип кнопки – **BitBtn**, со вкладки «Additional» (рис. 12). Изображение на такой кнопке задается через свойство **Glyph**. При этом необходимо заранее подготовить файл с этим изображением в формате **BMP**. Это можно сделать, например, при помощи программы **Paint**.

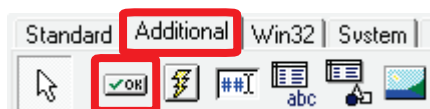


Рисунок 12 – Кнопка с изображением

Сохранение проекта

Для сохранения проекта, необходимо:

- выбрать меню «File» → «Save All» (рис. 13);

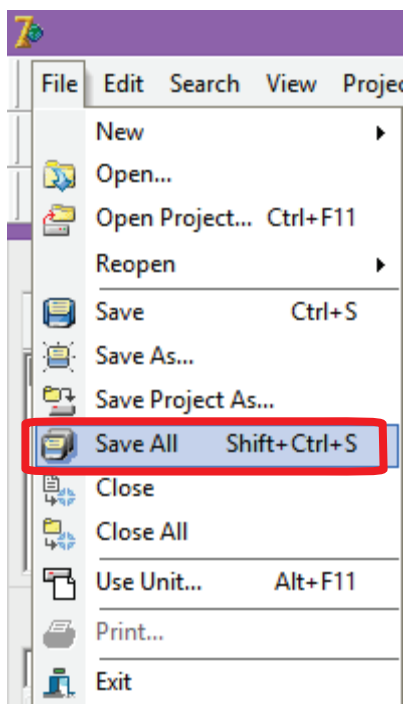


Рисунок 13 – Начало сохранения проекта

- создать папку; каждый проект нужно сохранять в отдельную папку! Например, назовем данную папку «Калькулятор»;
- сохранить **Unit** («Модуль») в эту папку (рис. 14); *имя модуля можно оставить «Unit1.pas»;*

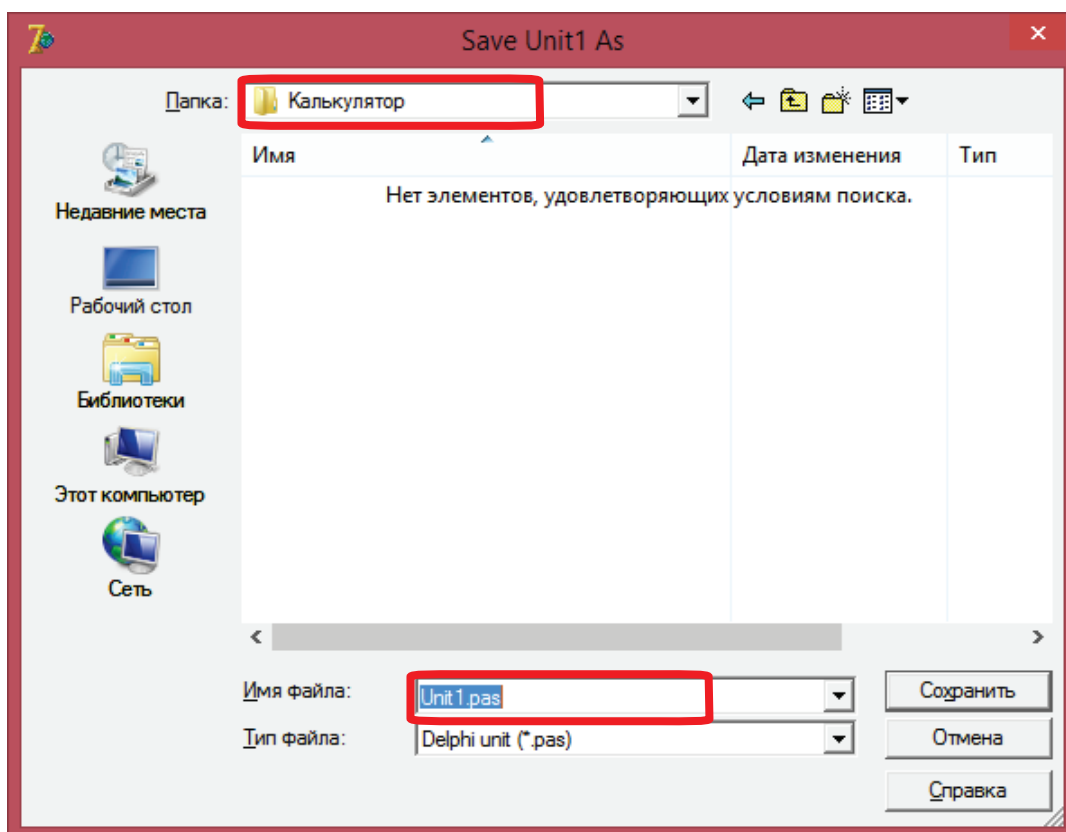


Рисунок 14 – Сохранение Модуля

- сохранить **Project** («Проект») в ту же папку (рис. 15). Для проекта необходимо придумать имя! Например, для калькулятора это может быть «**Calc**».

!!! Имена файлов должны состоять только из латинских букв, символов подчеркивания и цифр. Причем имя не может начинаться с цифры. В именах файлов не должно содержаться русских букв и пробелов!

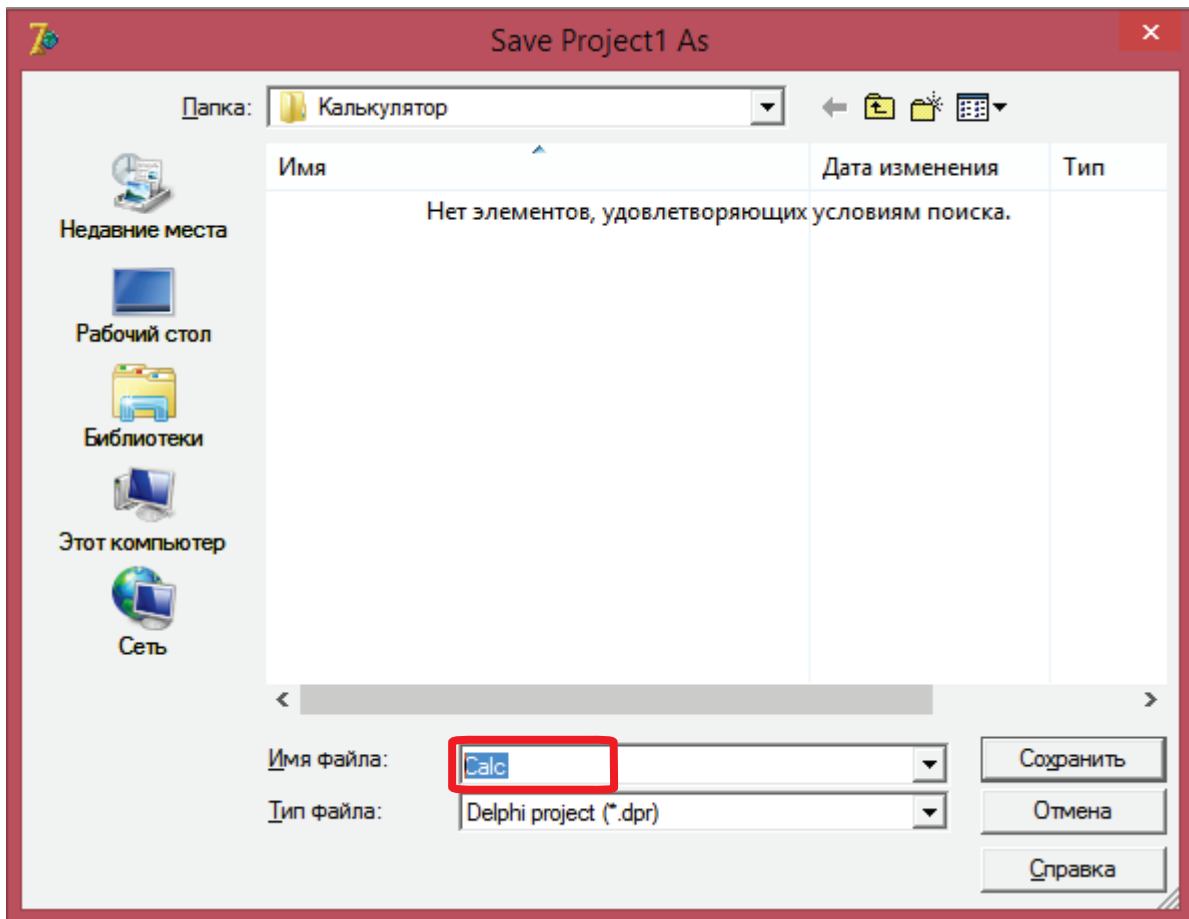


Рисунок 15 – Сохранение Проекта

Код программы

Когда графическая часть программы готова, пришло время написать наш первый Код. Для этого необходимо дважды кликнуть мышкой по добавленной ранее кнопке. В результате этих действий, **Delphi** перейдет в окно написания кода (рис. 16), а каретка (мигающий курсор для ввода текста) окажется между словами **begin** и **end** («начало» и «конец»).

!!! Прежде чем набирать Код, сначала нужно сделать отступ в 2 пробела. Аналогично нужно делать для каждого **begin** и **end** – весь код, находящийся в них, должен быть сдвинут правее на 2 символа.

!!! Когда в программу будет добавлено несколько кнопок, необходимо будет дважды кликнуть по каждой из них, и для каждой написать свой код.

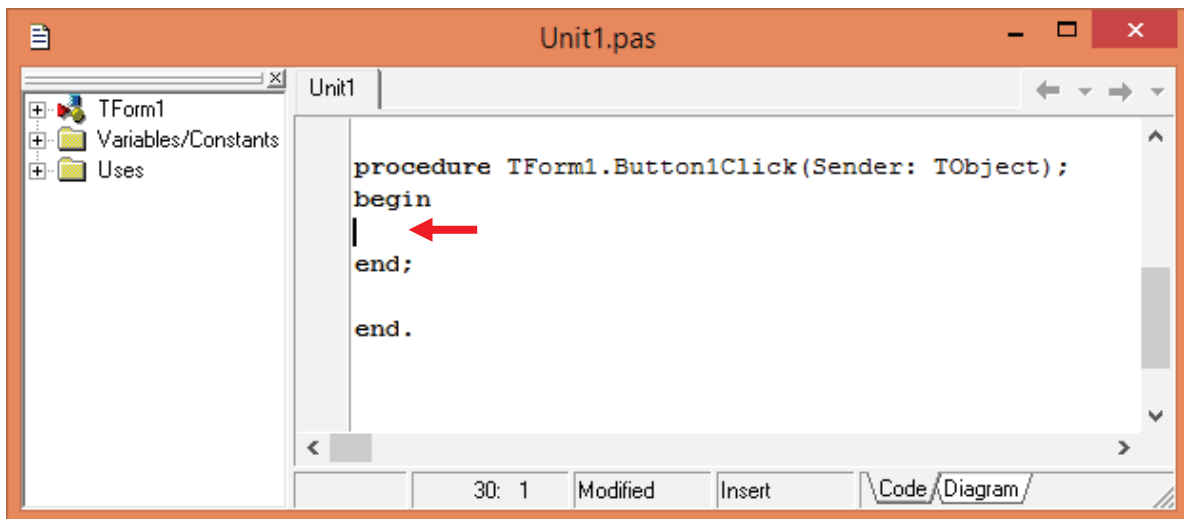


Рисунок 16 – Место для написания кода программы

У нас на Форме расположены два поля ввода (Edit1 и Edit2). Для обращения к тексту, введенному в них, нужно писать имена переменных как **Edit1.Text** и **Edit2.Text** (через точку!).

Напишем для начала следующий упрощенный код:

```

begin
    Edit2.Text := Edit1.Text;
end;

```

Если сейчас запустить программу (см. далее раздел 1.5), то этот код при каждом нажатии кнопки переносит в Edit2 значение, введенное в Edit1.

Но мы хотим, чтобы значение увеличивалось на 5, для этого необходимо написать код немного посложнее:

```

begin
    Edit2.Text := IntToStr(StrToInt(Edit1.Text) + 5);
end;

```

Здесь использовано преобразование типов (**IntToStr** и **StrToInt**), о котором будет рассказано чуть далее в разделе 1.7.

В конце каждой команды в языке **Паскаль** ставится точка с запятой.

!!! Паскаль допускает отсутствие точки с запятой в конце последней из команд. Но в студенческих работах использование точки с запятой является обязательным в конце каждой команды, в том числе последней!

Переключение между Формой и Кодом

Переключение между окнами редактирования Кода и редактирования Формы осуществляется кнопкой «**Toggle Form/Unit**» (рис. 17) или нажатием клавиши **F12** на клавиатуре.

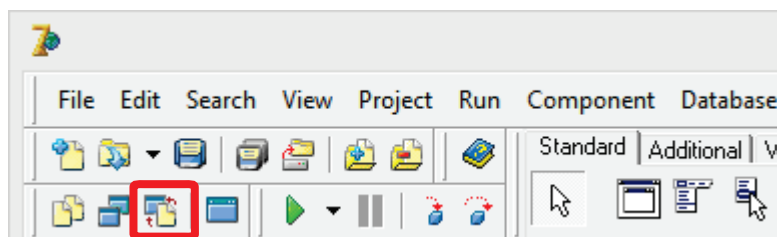


Рисунок 17 – Переключение между формой и кодом

1.5. Запуск программы и возможные ошибки

Когда программа готова, остается ее только запустить.

Запуск программы

Для запуска программы необходимо нажать кнопку «**Run**» (рис. 18) или нажать клавишу **F9** на клавиатуре. При этом если в программе имеются ошибки, то запуска не произойдет, и сначала нужно будет исправить эти ошибки.

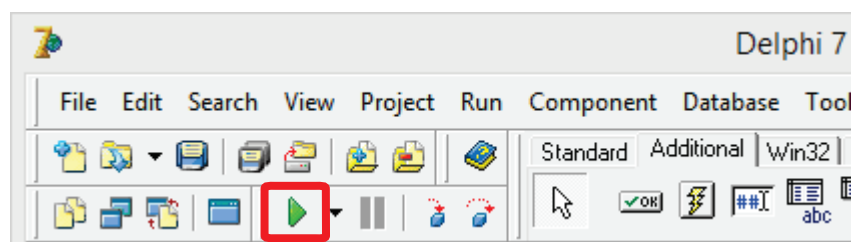


Рисунок 18 – Кнопка запуска программы

Ошибки компиляции

Если в коде имеются ошибки, то вместо запуска программы, внизу окна с кодом будет отображено сообщение об ошибке (рис. 19). В этом окне кратко будет описана ошибка, а если кликнуть дважды по этому сообщению, то в коде будет подсвечена строка, которую нужно исправить. После исправления программы нужно снова нажать «**Run**» (**F9**).

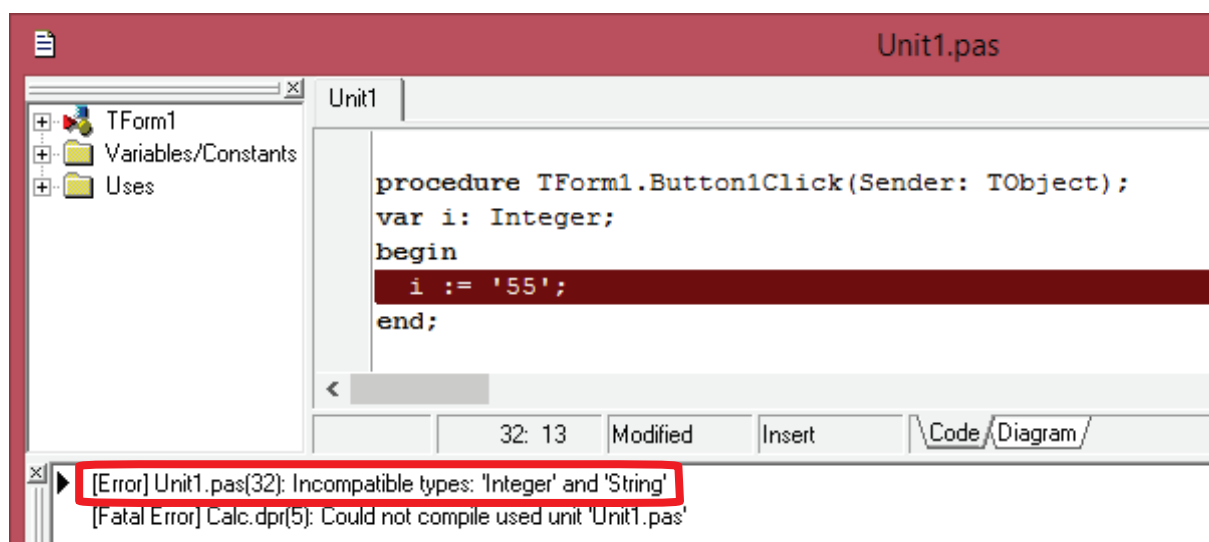


Рисунок 19 – Ошибка компиляции

Ошибки при работе программы

Ошибки могут возникать не только при компиляции, но и во время работы запущенной программы. Например, это может быть ошибка при попытке делить на ноль, или ошибка при преобразовании строки в число (рис. 20).

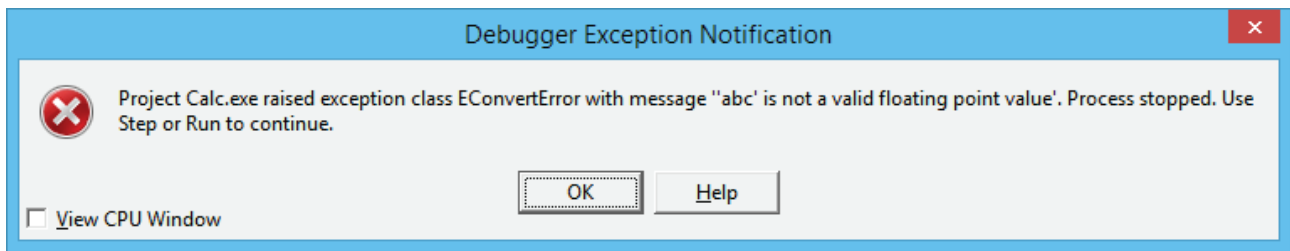


Рисунок 20 – Ошибка при работе программы

Необходимо нажать «**ОК**» в данном окне, и далее у нас есть два варианта:

- завершить работу программы. Для этого необходимо выбрать меню «Run» → «**Program Reset**» (рис. 21);

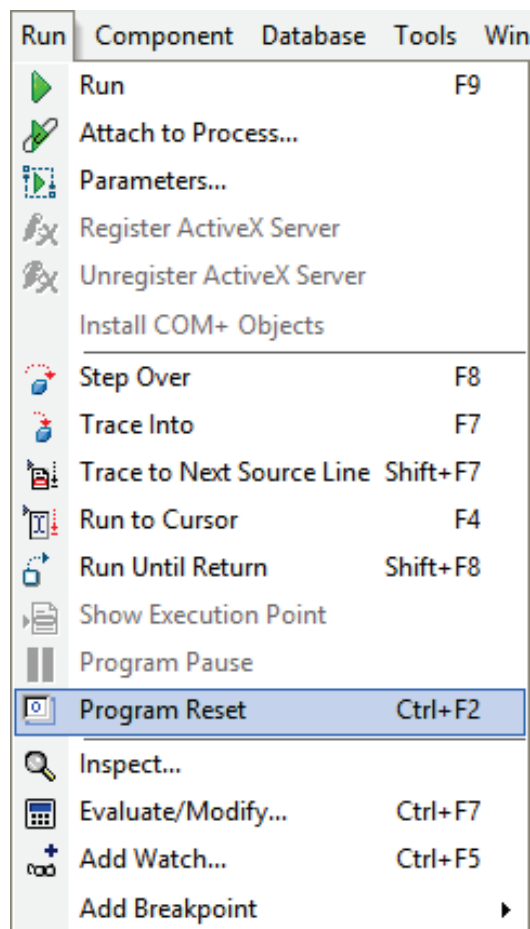


Рисунок 21 – Завершение работы программы

- или продолжить работу программы. Для этого необходимо повторно нажать «**Run**» (**F9**), после чего согласиться с сообщением об ошибке.

1.6. Объявление переменных и констант

!!! Теперь мы знаем, как запустить *Delphi*, поэтому весь описанный дальше код, лучше сразу протестировать с его помощью.

Значения можно записывать в переменную через оператор присваивания, например:

```
x := 5;
y := Sin(x) - 0.5;
```

Но сначала эти переменные нужно объявить. Имена переменных и констант не должны содержать русских букв и пробелов. Они могут состоять только из латинских букв, цифр и символов подчеркивания. Причем имя не может начинаться с цифры.

Кроме того, нельзя использовать в качестве имен переменных ключевые слова, зарезервированные в Паскале, такие как: **not, and, or, xor, div, mod, shl, shr, begin, end, procedure, function, var, const, type, unit, library, record, class, if, then, else, for, while, do** и др.

Локальные переменные

Локальные переменные в языке **Pascal** объявляются между строками «**procedure**» (либо «**function**») и «**begin**».

Ранее (после двойного клика по кнопке) у нас уже создавалась процедура, имеющая вид:

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

Для объявления переменных, процедуру нужно модифицировать следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var x, y: Integer;
    r: Real;
begin
    ...
end;
```

Объявление переменных всегда начинается с ключевого слова «**var**» («**Переменная**»). Допускается объявлять сразу несколько переменных одинакового типа, в этом случае они перечисляются через запятую. Можно объявлять и переменные разных типов, в этом случае они записываются в разных строках. Строки записываются через точку с запятой. В таком случае не нужно несколько раз писать ключевое слово «**var**».

С точки зрения оформления, каждая следующая строка должна быть выровнена по именам предыдущей (для **var** каждая следующая строка начинается с 4-х пробелов).

Локальные переменные видны только внутри соответствующей процедуры/функции (т.е. только между соответствующими **begin** и **end**, следующими за этим **var**).

В разных процедурах/функциях можно использовать одни и те же имена переменных, причем это будут разные, не зависящие друг от друга переменные. Значение локальной переменной теряется при выходе из процедуры/функции (т.е. когда программа доходит до соответствующего **end**).

Глобальные переменные

Глобальные переменные объявляются за пределами процедур (или функций), например:

```
var x, y: Integer;  
    r: Real;
```

```
procedure ...  
begin  
    ...  
end;
```

```
procedure ...  
begin  
    ...  
end;
```

Такие переменные видны в любой из процедур, расположенных ниже объявления переменных. Не следует использовать одинаковые имена для Локальных и Глобальных переменных.

!!! По умолчанию все переменные должны объявляться только как Локальные! При использовании Глобальных переменных автор должен доказать необходимость их применения.

Для Глобальных переменных доступно объявление начального значения (которого нет для Локальных переменных), например:

```
var x: Integer = 0;  
    r: Real = 5.2;
```

В таком случае можно объявлять только по одной переменной в строке (т.е. перечисление через запятую, в этом случае больше недоступно).

Объявление констант

Константы объявляются аналогично переменным, но начинаются с ключевого слова «**const**». Они также могут быть Локальными или Глобальными. Для констант наличие знака равенства и наличие значения обязательны, например:

```
const a = 5;  
      s = 'Привет!';
```

Константы (как и переменные) можно использовать справа от знака присваивания:

```
procedure ...  
const a = 5;  
var x: Integer;  
begin  
    x := a + 2;  
    ...  
end;
```

Но их нельзя использовать слева от знака присваивания. Это вызовет ошибку (рис. 22), т.к. нельзя записать в константу новое значение.

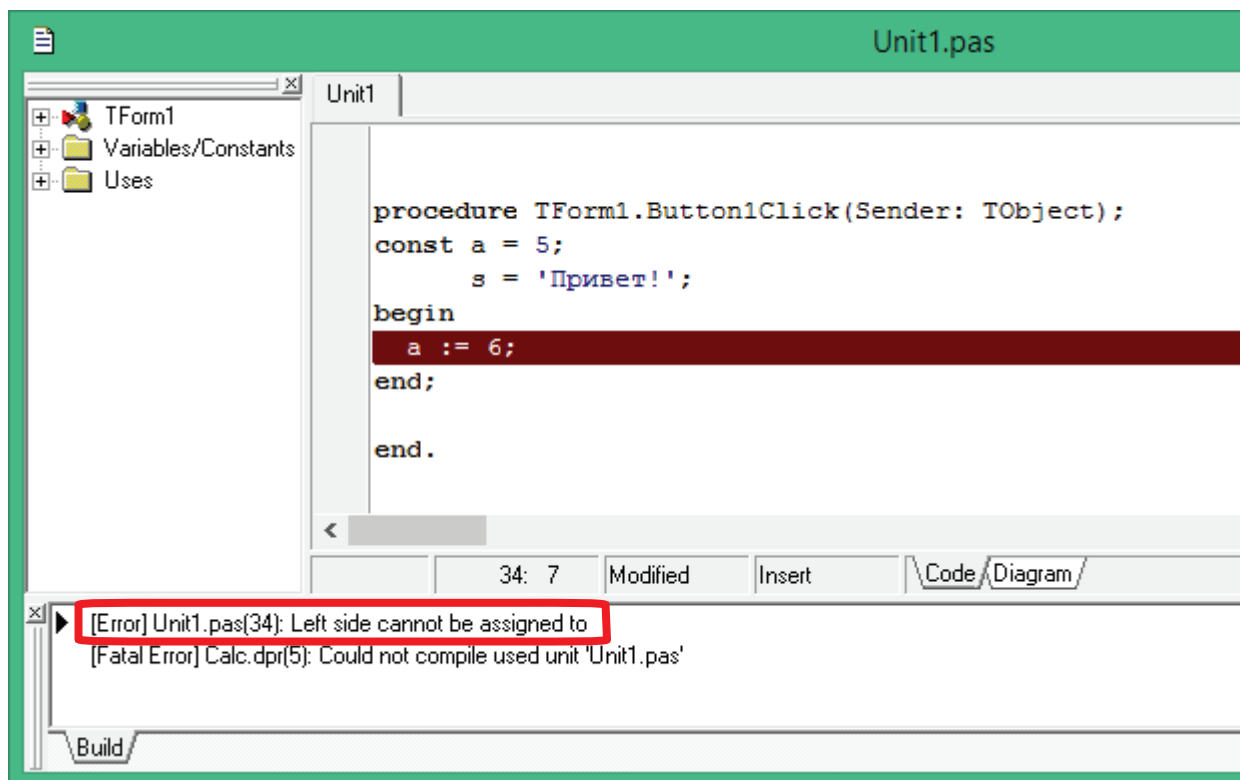


Рисунок 22 – Ошибка присваивания

Операторы в объявлениях переменных и констант

При объявлении начальных значений переменных, а также при объявлении констант, можно использовать операторы (сложение, вычитание, умножение, конкатенация и др.). Например:

```
const a = 7 - 3*2;
var S: String = 'Привет' + '!!!!';
```

Кроме того, здесь можно использовать имена констант (которые объявлены в программе раньше), например:

```
const a = 7 - 3*2;
b = 3 + a/2;
var R: Real = b;
```

Запись константных значений

Константам не обязательно давать имена, часто константные значения используются прямо в тексте программы.

Независимо от места использования, константы имеют следующие варианты записи значений:

- целые числа, положительные и отрицательные: 5, +5, -7;
- «нецелые» (вещественные) числа, записываемые через точку (не через запятую!): 3.5, +3.5, -7.7;
- числа, записанные в научном формате: 1.2e-5, -3.7E33;
- шестнадцатеричная запись целых чисел: \$FF00;
- логические значения: True, False;
- строковые значения, записываемые в одинарных кавычках: 'Привет';
- строковые символы: #9, #13#10.

1.7. Преобразование типов

Строковые значения записываются в кавычках, например, '55'. В свою очередь, числовые значения записываются без кавычек, например, 55. Несмотря на кажущееся сходство, это совершенно разные значения, и если попытаться записать число в строку, то мы получим ошибку (рис. 23).

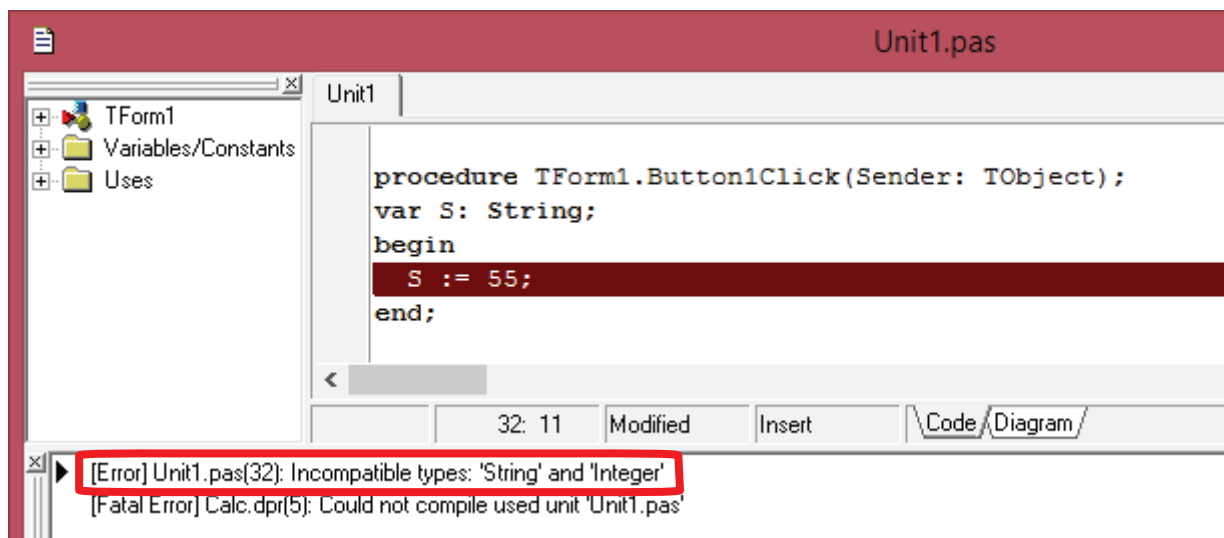


Рисунок 23 – Ошибка записи числа в строку

Также мы получим ошибку (рис. 24) и при попытке записи строки в число.

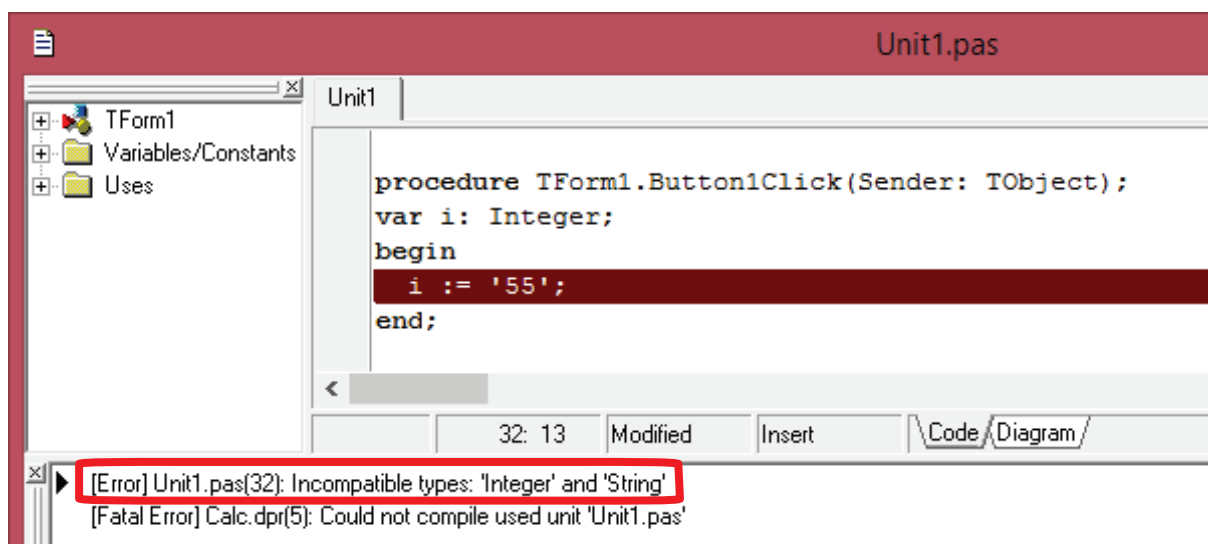


Рисунок 24 – Ошибка записи строки в число

Для преобразования типов (приведения типов) используются специальные функции. *Говоря упрощенно, эти функции позволяют добавлять и убирать кавычки.*

Преобразование в строку

Функции для преобразования значений в строку текста, представлены в таблице 12.

Таблица 12 – Функции преобразования в строку

Функция	Описание	Пример
IntToStr(I)	Преобразует целое число в строку	S := IntToStr(5); //S='5'
FloatToStr(R)	Преобразует «нецелое» число в строку	S := FloatToStr(2.75); //S='2,75'
BoolToStr(B)	Преобразует логическое значение в строку (результат '0' или '-1')	S1 := BoolToStr(False); //S1='0' S2 := BoolToStr(True); //S2='-1'
BoolToStr(B, True)	Преобразует логическое значение в строку (результат 'True' или 'False')	S3 := BoolToStr(False, True); //S3='False' S4 := BoolToStr(True, True); //S4='True'
IntToHex(I, Digits);	Преобразует целое число к его 16-ричному представлению. Дополнительный параметр Digits указывает минимальное количество 16-ричных разрядов в результате. Если результат короче, он дополняется нулями в начале	S := IntToHex(266, 4); //S='010A'

Преобразование из строки

Функции для преобразования строки текста в значения, представлены в таблице 13.

Таблица 13 – Функции преобразования из строки

Функция	Описание	Пример
StrToInt(S)	Преобразует строку в целое число	I1 := StrToInt('123'); //I1=123 I2 := StrToInt('123,45'); //Выдаст ошибку I3 := StrToInt('abc'); //Выдаст ошибку
StrToFloat(S)	Преобразует строку в «нецелое» число	R1 := StrToFloat('123,45'); //R1=123.45 R2 := StrToFloat('123E-3'); //R2=0.123 R3 := StrToFloat('abc'); //Выдаст ошибку

Функция	Описание	Пример
StrToBool(S)	Преобразует строку в логическое значение	<pre> B1 := StrToBool('0'); //B1=False B2 := StrToBool('1'); //B2=True B3 := StrToBool('true'); //B3=True B4 := StrToBool('FaLsE'); //B4=False B5 := StrToBool('abc'); //Выдаст ошибку B6 := StrToBool('-1'); //B6=True B7 := StrToBool('100'); //B7=True </pre>

1.8. Математические функции

Параметры (аргументы) функций указываются в скобках после имени функции. Если их несколько, то они перечисляются через запятую.

Основные математические функции представлены в таблице 14.

Таблица 14 – Основные математические функции

Функция	Описание	Пример
Abs(R)	Модуль числа (<i>абсолютное значение</i>)	<code>R := Abs(-5); //R=5</code>
Pi	Число π (<i>константа</i>)	<code>R := Pi; //R=3.14159...</code>
Sqr(R)	Квадрат числа ($R^2=R*R$)	<code>R := Sqr(9); //R=81</code>
Sqrt(R)	Квадратный корень числа (\sqrt{R})	<code>R := Sqrt(9); //R=3</code>
Power(Base, Exp)	Степень числа ($Base^{Exp}$)	<code>R := Power(5, 3); //R=125</code>
Random(I)	Генерирует случайное целое число в диапазоне от 0 до (I-1). Перед использованием нужно выполнить команду Randomize	<pre> Randomize; I := Random(100); {I=37, или любое др. число в диапазоне 0..99} </pre>
Log10(R)	Логарифм по основанию 10	<pre> R1 := Log10(100); //R1=2 R2 := Log10(1000); //R2=3 R3 := Log10(3000); //R3=3.47... </pre>
Ln(R)	Натуральный логарифм (т.е. логарифм по основанию $e=2,718...$)	<pre> R1 := Ln(10); //R1=2.302585... R2 := Ln(2.718282); //R2=1.00... </pre>
Log2(R)	Логарифм по основанию 2	<code>R := Log2(32); //R=5</code>
LogN(N, R)	Логарифм по заданному основанию N	<code>R := LogN(3, 81); //R=4</code>

Функция	Описание	Пример
Exp (R)	Экспонента числа ($e^R=2,71828^R$)	R1 := Exp(3); //R1=20.0855... R2 := Exp(1); //2.718...
Max (R1, R2) Min (R1, R2)	Максимум и минимум из двух чисел	R1 := Max(10, 7); //R1=10 R2 := Min(10, 7); //R2=7
Sin (R) Cos (R) Tan (R) Cotan (R)	Тригонометрические функции (синус, косинус, тангенс, котангенс)	R1 := Sin(0); //R1=0 R2 := Cos(0); //R2=1 R3 := Sin(90); //R3=0.893... R4 := Sin(90*Pi/180); //R4=1
ArcSin (R) ArcCos (R) ArcTan (R)	Обратные тригонометрические функции (арксинус, арккосинус, арктангенс)	R5 := ArcSin(1/2); //R5=0.523... R6 := ArcSin(0.5)*180/Pi; //R6=30 – в градусах
Sinh (R) Cosh (R) Tanh (R) ArcSinh (R) ArcCosh (R) ArcTanh (R)	Гиперболические функции и обратные гиперболические функции	R := Sinh(1); //R=1.1752...
Round (R)	Округление до целого в ближайшую сторону (т.е. по правилу округления)	I1 := Round(7.5); //I1=8 I2 := Round(7.4); //I2=7
Trunc (R) Floor (R) Ceil (R)	Варианты округления до целого в меньшую или в большую сторону	I := Trunc(7.9); //I=7

Некоторые из перечисленных функций описаны в дополнительном модуле «**Math**». Для их использования необходимо прописать команду подключения модуля:

uses Math;

!!! При этом в программе по умолчанию уже объявлен блок «**uses**», повторно его использовать не нужно! Нужно промотать Код в самое начало и через запятую дописать к списку новый модуль (рис. 25).

```

Unit1
unit Unit1;

interface

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

```

Рисунок 25 – Подключение модуля

1.9. РАБОТА № 1

«Разработка программы «Калькулятор»»

Задание

Разработать программу-калькулятор, похожую на калькулятор Windows (в режиме «Инженерный», с некоторыми дополнениями из режима «Программист»).

Пример внешнего вида программы «Калькулятор» представлен на рисунке 26. Положение кнопок и надписи на кнопках могут отличаться. Специальные символы (такие как « \pm ») можно найти в «таблице символов».

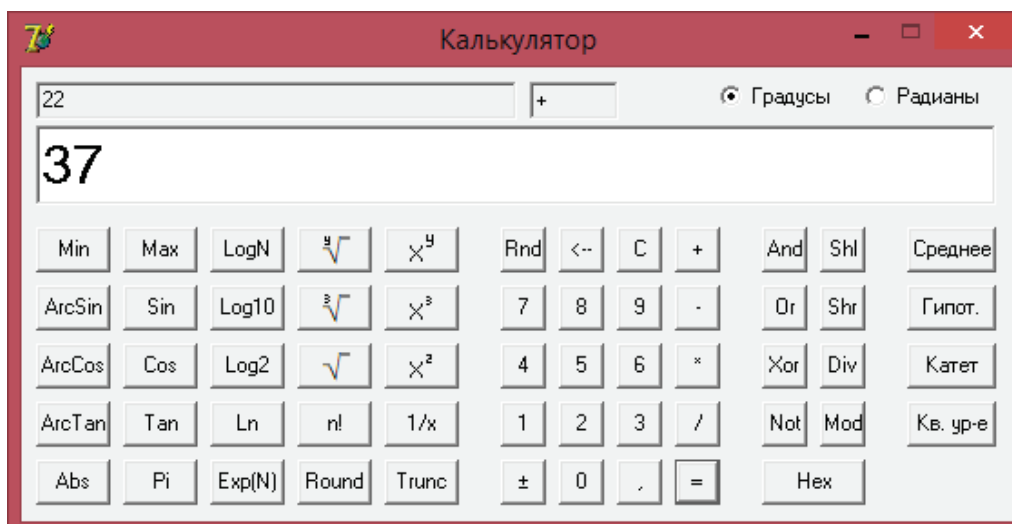


Рисунок 26 – Внешний вид программы «Калькулятор»

Работа с углами должна производиться в градусах (либо в градусах и радианах на выбор, как показано в верхнем правом углу на рисунке 26).

Функции программы

В центральной секции:

- ввод цифр от «0» до «9» и запятой «,»;
- изменение знака числа (« \pm »);
- кнопка «C» очищающая все поля и кнопка « \leftarrow » стирающая последний символ;
- кнопка «Rnd», выводящая случайное число (от 0 до 999);
- арифметические операции «+», «-», «*» и «/»;
- кнопка «=», вычисляющая результат (только для операторов и функций с двумя аргументами).

Левая секция:

- корни и степени;
- логарифмы и экспонента;
- факториал («n!»);
- округление («Round» и «Trunc»);
- минимальное и максимальное значение из двух чисел («Min» и «Max»);
- тригонометрические функции, обратные тригонометрические функции и число π ;

- модуль числа («Abs»).

Правая секция (целочисленные операции):

- побитовые логические операции («And», «Or», «Xor», «Not»);
- битовые сдвиги («Shl», «Shr»);
- целочисленное деление и остаток от деления («Div», «Mod»);
- перевод числа в 16-ричный вид («Hex»).

Дополнительная секция с пользовательскими функциями:

- расчет среднего для двух чисел;
- вычисление гипотенузы по двум катетам;
- вычисление катета по гипотенузе и второму катету;
- решение квадратного уравнения вида $X^2 + b \cdot X + c = 0$ по заданным значениям b и c .

Дополнительные пользовательские функции должны быть вынесены в отдельный модуль (библиотеку функций). *Подробнее про создание собственных функций и собственных модулей будет рассказано далее (в разделах 1.12 и 1.13).*

Пояснения по работе программы

Помимо кнопок, программа состоит из трех полей Edit1, Edit2 и Edit3 (рис. 27).

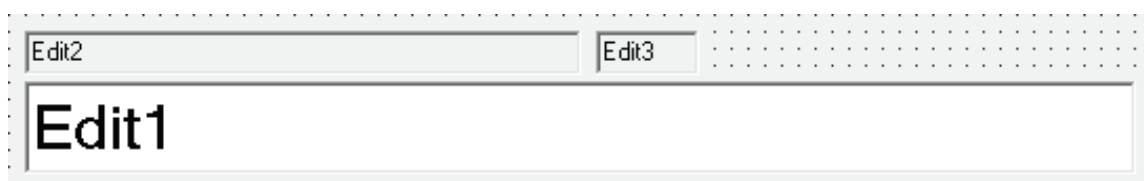


Рисунок 27 – Поля программы

Все задействованные операторы и функций можно разделить на две группы:

1. С одним входным аргументом (или вообще без аргументов), такие, например, как: Sin, Abs, Rnd, X^2 , Pi, Not и т.п. Для вычисления этих значений не применяется кнопка «=», результат сразу помещается в Edit1. Другие поля (Edit2, Edit3) в этом случае не используются.
2. С двумя входными аргументами, такие, например, как: +, Max, $\sqrt[X]{}$, Xor и т.п. Результат в этом случае вычисляется после нажатия кнопки «=», а ввод исходных данных производится в два этапа. На первом этапе вводится значение первого из аргументов, после чего нажимается кнопка с требуемой операцией (например, «Max»). В результате этих действий, первый аргумент переносится из Edit1 в Edit2, а операция («max») запоминается в Edit3. Последовательность работы на первом этапе продемонстрирована на рисунке 28. На втором этапе вводится значение второго аргумента и нажимается кнопка «=». В результате этих действий происходит вычисление функции или оператора, указанного в Edit3 (с аргументами, указанными в Edit1 и Edit2). Результат помещается в Edit1.

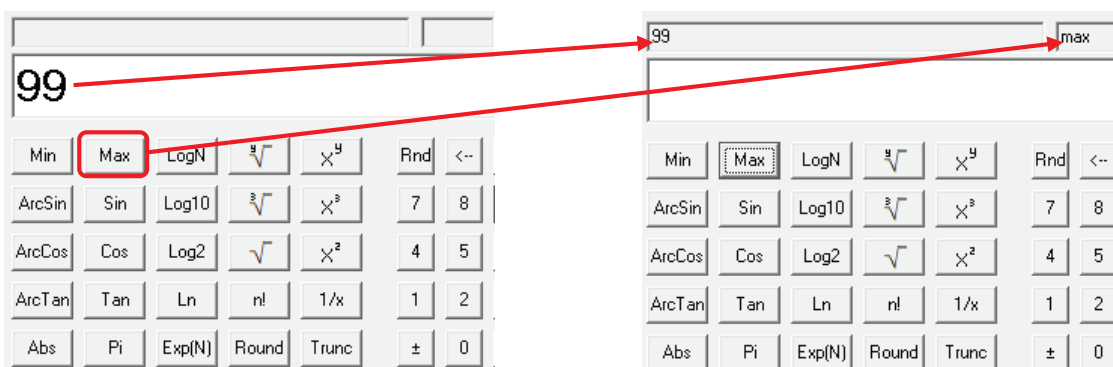


Рисунок 28 – Ввод первого аргумента

Отчет должен содержать:

- краткое Задание (не более 1-ой стр.). В задании также должен быть отражен вариант, который применялся для работы с углами в данном случае («в градусах», или «имеет выбор градусы или радианы»). Если автор делал программу с какими-либо дополнениями, изменениями и усложнениями, то это должно быть отражено в задании;
- скриншоты получившейся программы;
- список функций, выполняемых программой. Список функций может быть оформлен по-разному, в том числе, можно оформить его в виде таблицы. Можно разбить список на группы родственных функций, или не применять группировку. Но, в любом случае, функции должны легко соотноситься со скриншотами программы, например, для этого можно на скриншоте использовать сноски с номерами (1, 2, 3, ...), которые дальше использовать при составлении списка, либо не применять нумерацию на скриншоте, но оформить список функций в виде таблицы, вставив в эту таблицу отдельные изображения для каждой из кнопок, либо использовать др. вариант;
- код главной программы (с комментариями!). Желательно разбить код на две четко выраженные части, одна для функций с одним аргументом, вторая – для функций с двумя аргументами;
- код модуля (с комментариями!) и его краткое описание (Что делают данные функции? Как получили формулы, по которым их рассчитываем? В каких случаях сколько корней имеет уравнение? Что такое гипотенуза? и т.п.);
- титульный лист, номера страниц, оглавление, список использованной литературы (включая Интернет-ресурсы и данное пособие), выводы/заключение и т.п.

1.10. *РАБОТА № 1 (дополнение) «Калькулятор с двумя полями»

Требуется модернизировать предыдущее задание таким образом, чтобы не использовалось третье поле Edit3, в котором указывается операция. В этом случае операция приписывается в конце поля Edit2. Последовательность

работы на первом этапе для данной программы продемонстрирована на рисунке 29.

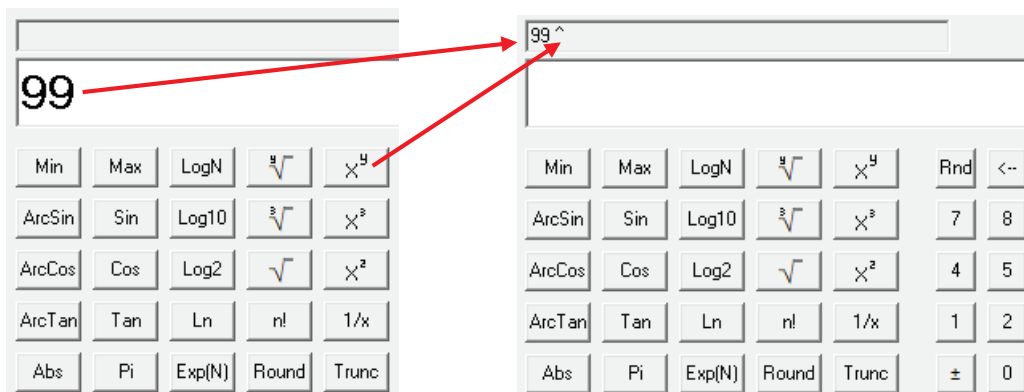


Рисунок 29 – Вариант с двумя полями

На втором этапе (после нажатия кнопки « \Leftarrow ») операция вырезается из Edit2, и из этого же Edit2 берется первый аргумент.

Данная модернизация не влияет на процесс вычисления операторов и функций с одним аргументом (или вообще без аргументов).

1.11. Строковые функции

Основные функции для работы с текстовыми строками представлены в табл. 15.

Таблица 15 – Основные строковые функции

Функция	Описание	Пример
UpperCase(S), AnsiUpperCase(S)	Преобразует строку к верхнему регистру. Версия Ansi позволяет работать с русскими символами	<code>S := AnsiUpperCase('TeKcT');</code> <code>//S='ТЕКСТ'</code>
LowerCase(S), AnsiLowerCase(S)	Преобразует строку к нижнему регистру	<code>S := AnsiLowerCase('TeKcT');</code> <code>//S='текст'</code>
Trim(S), TrimLeft(S), TrimRight(S)	Удаляет из строки начальные и завершающие пробелы (и др. специальные символы, такие как табуляция). Версии Left и Right удаляют, соответственно, только начальные или только завершающие символы	<code>I1 := StrToInt(' 123 ');</code> <code>//Выдаст ошибку</code> <code>I2 := StrToInt(Trim(' 123 '));</code> <code>//I2=123</code>

Функция	Описание	Пример
Copy (S, Index, Count)	Возвращает подстроку строки S , начиная с символа с номером Index , содержащую до Count символов. <i>Нумерация символов в строках начинается с единицы</i>	S := '1234567890'; S := Copy(S, 3, 5); //S='34567'
Length (S)	Длина строки (количество символов в строке)	I := Length('Привет!'); //I=7
Pos (Substr, S)	Возвращает позицию (индекс) первого вхождения подстроки Substr в строку S . Если S не содержит Substr , то возвращает 0	S := '012345678'; I1 := Pos('9', S); //I1=0 I2 := Pos('34', S); //I2=4
StringReplace (S, Old, New, []);	Заменяет в строке S <u>первое</u> вхождение Old на New	S := 'a=12;b=1;c=7;d=21'; S := StringReplace(S, '1', '2', []); //S='a=22;b=1;c=7;d=21'
StringReplace (S, Old, New, [rfReplaceAll]);	Заменяет в строке S <u>все</u> вхождения Old на New	S := 'a=12;b=1;c=7;d=21'; S := StringReplace(S, '1', '2', [rfReplaceAll]); //S='a=22;b=2;c=7;d=22'
FormatFloat (Format, R)	Округление и формат отображения чисел с плавающей точкой (запятой). Символ '0' – означает любую цифру, включая 0; Символ '#' – любая цифра (кроме нуля) или ничего; Символ '.' – положение десятичного разделителя; Символ ',' – пробел при отображении тысяч (групп разрядов по 3 символа); Символ ';' – разделитель форматов для положительных, отрицательных и нулевых значений; и др.	R1 := 12.34; R2 := 0.6789; S1 := FormatFloat('#.##0', R1); //S1='12,340' S2 := FormatFloat('#.##0', R2); //S2=',679' S3 := FormatFloat('0.###', R1); //S3='12,34' S4 := FormatFloat('0.###', R2); //S4='0,679' R3 := 12347.55; S5 := FormatFloat('#,##0.###', R3); //S5='12 347,55'

Функция	Описание	Пример
Chr(I)	Получение символа по его коду	S1 := Chr(9); //Символ табуляции S2 := Chr(13) + Chr(10); //Переход на новую строку //(Enter) S3 := Chr(68); //Буква 'D'

Задача 1

В Edit1 пользователем вводится строка текста. Преобразовать ее к нижнему регистру, за исключением первого символа, который необходимо сделать большой буквой. Результат вывести обратно в Edit1.

!!! Задачи отличаются от Работ тем, что их необходимо выполнять не на компьютере, а в тетрадке. Для закрепления материала рекомендуется также проверять их дома на компьютере.

1.12. Создание собственных процедур и функций

Объявление процедур начинается с ключевого слова «*procedure*», и с ними мы уже сталкивались ранее, т.к. для каждой кнопки **Button** автоматически создавалась отдельная процедура.

Процедуры объявляются как:

```
procedure ProcName(Arg1: Integer; Arg2: String);
begin
    ...
end;
```

!!! Имена процедур и функций должны состоять только из латинских букв, символов подчеркивания и цифр. Причем имя не может начинаться с цифры. В именах не должно содержаться русских букв и пробелов!

Функции похожи на процедуры, но отличаются тем, что функция **всегда** возвращает какое-либо значение (**результат**). Тип возвращаемого значения указывается после скобок через двоеточие (а если скобок нет, то через двоеточие, сразу после имени). Пример объявления собственной функции:

```
function FuncName(Arg1: Integer; Arg2: String): Integer;
begin
    ...
    Result := ...
end;
```

Значение результата присваивается переменной **Result**. Использование переменной **Result**, т.е. возврат результата – обязателен!

Как для функций, так и для процедур, входные параметры (**аргументы**) записываются в скобках. Если имеется несколько аргументов, то они объявляются через точку с запятой. Аргументов может быть и ноль, тогда скобки не пишутся.

Несколько аргументов одного типа, идущих подряд, могут быть записаны через запятую. Например:

```
(Arg1, Arg2: Integer; Arg3: String; Arg4: Integer)
```

Либо, то же самое полностью:

(Arg1: Integer; Arg2: Integer; Arg3: String; Arg4: Integer)

Создадим собственную функцию **Avg**, рассчитывающую среднее из двух значений:

```
//Среднее значение
function Avg(A, B: Real): Real;
begin
  Result := (A + B) / 2;
end;
```

Эту функцию необходимо объявить в коде выше, чем она будет использоваться, например:

```
function Avg...
begin
  ...
end;

...

procedure TForm1.Button1Click...
begin
  ...
  R := Avg(5, 4.5);
  ...
end;
```

Также можно вынести функцию в отдельный Модуль, о чем будет рассказано далее в разделе **1.13**.

Задача 2

Строка S содержит значения параметров, записанных в виде:

```
S := 'a=3;b=7;c=5;d=1';
```

В этом упрощенном примере будем рассматривать вариант, при котором имена параметров состоят только из одной буквы, а их значения только из одной цифры.

Разработать функцию **GetVal**, возвращающую числовое значение для параметра с заданным именем **Name**.

Примеры использования функции:

```
R := GetVal(S, 'b'); //R=7
```

```
R := GetVal(S, 'c'); //R=5
```

Задача 3

Доработать предыдущую задачу до случая, при котором:

- имена параметров могут состоять из нескольких символов;
- значения могут быть нецелыми числами, в том числе отрицательными;
- нецелые числа могут записываться одновременно как через запятую, так и через точку;
- исходная строка может содержать пробелы в произвольных местах, при этом они игнорируются;
- точка с запятой после последнего значения является необязательной (может ставиться, а может не ставиться).

Пример входной строки:

```
S := ' a=3; bb =4.5 ;b= 5,7; cc= -8.2 ';
```

Пояснения для «Калькулятора»

При разработке «Калькулятора» необходимо создать собственные функции для расчета среднего для двух чисел, катета, гипотенузы и нахождения корней квадратного уравнения. Функцию для нахождения среднего мы уже получили чуть выше, теперь рассмотрим оставшиеся.

Вычисление катета и гипотенузы основаны на прямоугольном треугольнике (рис. 30).

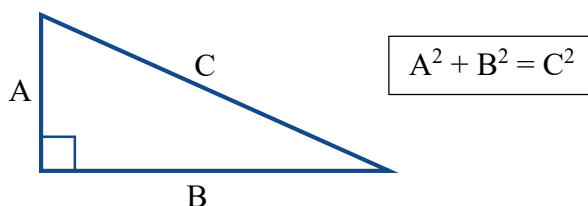


Рисунок 30 – Прямоугольный треугольник

В одном случае нам нужно найти **C** (по заданным **A** и **B**), а в другом **A** (по заданным **B** и **C**).

!!! Стоит обратить внимание, что в одном из вариантов под корнем возникает разность двух чисел, а это значит, что подкоренное выражение может стать отрицательным, что, в свою очередь, приведет к ошибке. Но при этом функция должна позволять вводить аргументы в любом порядке (как «больший»-«меньший», так и «меньший»-«больший»).

Для решения квадратного уравнения (вида $X^2 + b \cdot X + c = 0$) мы будем использовать результат в виде текстовой строки (**String**). У нас возможны 3 различных случая, например:

```
//X^2 + X + 1 = 0
```

```
S := Quadratic(1, 1); //S='Нет решений'
```

```
//X^2 + 2*X + 1 = 0
```

```
S := Quadratic(2, 1); //S='Один корень: X=-1'
```

```
//X^2 - 5*X + 6 = 0
```

```
S := Quadratic(-5, 6); //S='Два корня: X1=2, X2=3'
```

1.13. Модули Pascal

Модуль (unit) – это часть программы, сохраняемая в отдельный файл. В **Pascal** эти файлы имеют расширение ***.pas**.

Каждое окно (Форма) в **Delphi** сохраняется как отдельный модуль («оконный» модуль). *Т.е. мы уже встречались ранее с файлами с расширением *.pas*. Модули могут быть и без окон. Такие «безоконные» модули чаще всего используются как библиотеки (это могут быть библиотеки функций, библиотеки процедур или библиотеки компонентов).

Имя модуля всегда совпадает с именем **pas**-файла, и в коде прописывается автоматически при первом сохранении файла (*т.е. его не нужно менять в Коде вручную!*). Имя не должно содержать пробелов (но может содержать знак подчеркивания «_»). Имя модуля пишется латинскими буквами.

Структура модуля

Модуль имеет следующую структуру:

```
unit ИмяМодуля;  
  
interface  
{Блок Uses}  
{Блок объявления заголовков процедур и функций,  
 а также переменных, констант и типов, доступных для вызова}  
  
implementation  
{Блок Uses}  
{Реализация процедур и функций,  
 а также объявление переменных, констант и типов,  
 используемых только внутри модуля}  
  
initialization  
{...}  
  
finalization  
{...}  
  
end.
```

Модуль всегда состоит минимум из двух секций – «**interface**» и «**implementation**».

В секции «**interface**» описывается интерфейс взаимодействия модуля с внешним миром, т.е. то, что будет видно вызывающему модулю. Здесь содержится объявление заголовков процедур и функций. В этой секции не пишется Код!

Секция «**implementation**» содержит «реализацию» модуля, т.е. Код для процедур и функций пишется в этой секции. Эта часть модуля закрыта для внешнего мира (если мы только явно не укажем заголовки от процедур и функций, объявленных здесь, в предыдущей секции «**interface**»).

Внутри каждой из секций «**interface**» и «**implementation**» возможно подключение других модулей (**uses**), объявление переменных, констант и типов (**var, const, type**), объявление процедуры и функции (**procedure, function**). Но отличие в том, что в «**interface**» объявляются только заголовки от процедур и функций, а их реализация (т.е. Код, который пишется между «**begin**» и «**end**») может быть записана только в секции «**implementation**».

Секции «**initialization**» и «**finalization**» содержат Код, который выполняется соответственно при запуске Модуля и завершении его работы. Эти секции не являются обязательными. *Далее мы не будем их применять.*

Модуль всегда заканчивается словом «**end**» с точкой на конце. В одном модуле можно использовать неограниченное количество функций, процедур и компонентов. В том числе их можно смешивать в одном модуле.

Создание библиотеки функций

Создадим библиотеку функций для **Работы №1** («Калькулятор»), содержащую функцию **Avg**. Для этого необходимо открыть созданный ранее Проект и выбрать меню «File» → «New» → «Unit» (рис. 31). После чего нужно нажать «Сохранить» и задать имя Модуля «MyLib.pas». Сохранять модуль обязательно нужно в ту же папку, куда ранее был сохранен Проект.

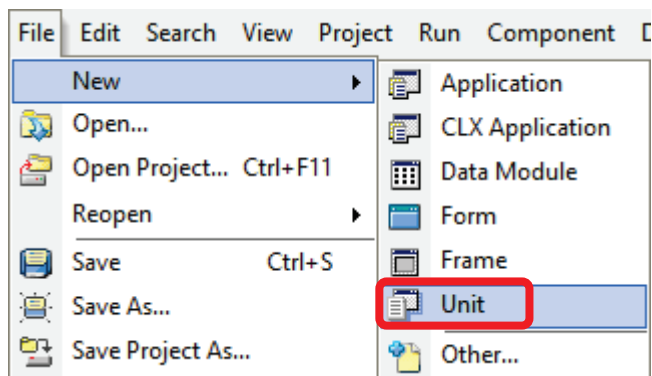


Рисунок 31 – Добавление нового модуля

Код нашего модуля будет следующим:

```
unit MyLib;  
  
interface  
  
    //Среднее значение  
    function Avg(A, B: Real): Real;  
  
implementation  
  
    //Среднее значение  
    function Avg(A, B: Real): Real;  
    begin  
        Result := (A + B) / 2;  
    end;  
  
end.
```

Для использования нового модуля, его требуется подключить в блоке **uses** вызывающего модуля (*того, который с окном*):

```
uses ..., MyLib;
```

Далее необходимо объявить заголовки для всех функций, которые будут входить в нашу библиотеку, например:

```
interface  
  
    //Среднее значение  
    function Avg(A, B: Real): Real;  
    //Нахождение гипотенузы по значениям двух катетов  
    function Hypotenuse(A, B: Real): Real;  
    //Нахождение катета по значениям гипотенузы и др. катета  
    function Katet(A, B: Real): Real;  
    //Решение квадратного уравнения вида  $x^2 + B*x + C = 0$   
    function Quadratic(B, C: Real): String;
```

После чего написать реализацию для этих функций.

Создание оконного модуля

Для добавления еще одного окна к программе необходимо выбрать меню «File» → «New» → «Form» (рис. 32). После чего нужно нажать «Сохранить» и задать имя для нового **pas**-файла. Сохранять оконный модуль обязательно нужно в ту же папку, куда ранее был сохранен Проект.

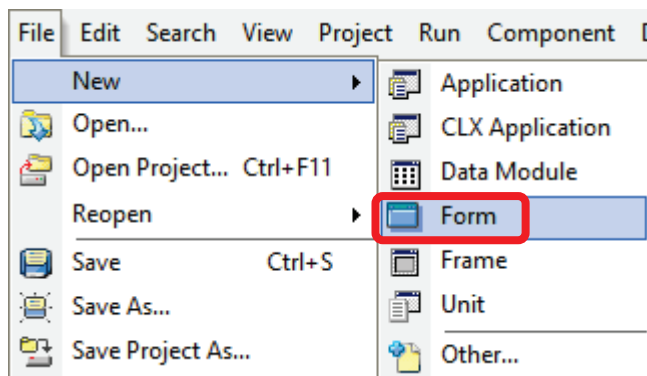


Рисунок 32 – Создание нового окна

Для использования нового окна, его необходимо подключить в блоке **uses** Главного окна.

Для вызова нового окна необходимо написать следующий код:

```
Form2.Show;
```

Если имя (**Name**) новой формы отличается, то вместо Form2 необходимо использовать соответствующее имя.

1.14. Некоторые полезные процедуры

Некоторые другие полезные процедуры и функции представлены в таблице 16.

Таблица 16 – Некоторые полезные процедуры и функции

Процедура	Описание
Beep;	Выдает звуковой сигнал «Бип»
Inc(I); Dec(I);	Инкремент и Декремент. Увеличивает или уменьшает значения на единицу. Равносильны: I := I + 1; I := I - 1;
ShowMessage(S);	Отображает окно с сообщением. Например: ShowMessage('Привет!');
Close; <i>или полностью:</i> Form1.Close; <i>где Form1 – имя окна, для которого выполняется команда</i>	Закрывает окно. <i>При закрытии Главного (или единственного) окна, закрывается и вся программа</i>

Процедура	Описание
Form2.Show; Form2.Hide; <i>где Form2 – имя окна, для которого выполняется команда</i>	Открыть (показать) или скрыть указанное окно

Пример окна с сообщением, полученным при помощи **ShowMessage**, представлен на рисунке 33.

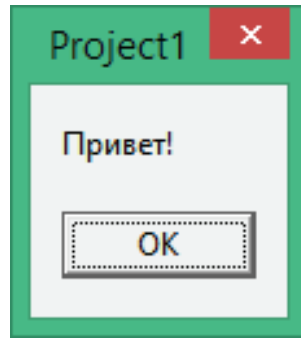


Рисунок 33 – Окно с сообщением

1.15. Условный оператор if

Ранее мы рассматривали только случаи, при которых все действия (после нажатия какой-либо Кнопки) выполняются всегда. Но бывают случаи, когда действия в разных ситуациях могут выполняться или не выполняться. Такие действия необходимо «обернуть» условным оператором.

Основным из условных операторов является оператор **if** («если»), имеющий несколько вариантов записи.

1. Однострочный

if условие **then** действие;

Читается как «если условие, то действие». В конце всегда ставится точка с запятой.

Например, рассмотрим следующую функцию:

```
function DivMod(A, D: Integer): String;
var M: Integer;
begin
  //Целая часть
  Result := IntToStr(A div D);
  //Остаток
  M := A mod D;
  Result := Result + ' ' + IntToStr(M) + '/' + IntToStr(D);
end;
```

Данная функция выполняет деление **A** на **D** и возвращает результат в виде целой и дробной частей. Примеры вызова данной функции следующие:

```
S := DivMod(7, 5); //S='1 2/5'
S := DivMod(15, 5); //S='3 0/5'
```

Но нет смысла выводить «0/5», поэтому мы хотим, чтобы при нулевом остатке, дробная часть не отображалась, т.е.:

```
S := DivMod(15, 5); //S='3'
```

Поэтому обернем последнюю команду условием **if**:

```
if M > 0 then Result := Result + ' ' + IntToStr(M) + '/' + IntToStr(D);
```

Целая часть также может быть нулевой, поэтому можно обернуть аналогичным условием и ее вывод.

2. Многострочный

Если действие всего одно, то **begin** и **end** использовать не обязательно, но если действий несколько, то они обязательно должны располагаться между **begin** и **end**:

```
if условие then
begin
действие_1;
действие_2;
...
end;
```

!!! Находясь внутри **begin** и **end**, мы всегда должны сделать отступ вправо (минимум на 2 пробела)!

3. Однострочный с двумя ветвями

```
if условие then действие_1 else действие_2;
```

Читается как «если условие – то действие 1, иначе – действие 2». В конце всегда ставится точка с запятой. При этом точка с запятой перед **else** (т.е. после «действие_1») не ставится!

4. Многострочный с двумя ветвями

```
if условие then
begin
действие_1_1;
действие_1_2;
...
end
else
begin
действие_2_1;
действие_2_2;
...
end;
```

Точка с запятой перед **else** (т.е. после первого **end**) не ставится! Внутри каждого **begin** и **end**, мы всегда должны делать отступы вправо.

Пояснения для «Калькулятора»

В «Калькуляторе» нажатие кнопки «=» выполняет различные действия в зависимости от того, какие данные мы ввели ранее (от того, какие кнопки мы ранее нажали).

Например, если ранее мы нажимали кнопку «+», то этот оператор был запомнен в поле **Edit3** (рис. 34). Соответственно, при нажатии «=» в этом случае выполнится сложение чисел из полей **Edit1** и **Edit2**.

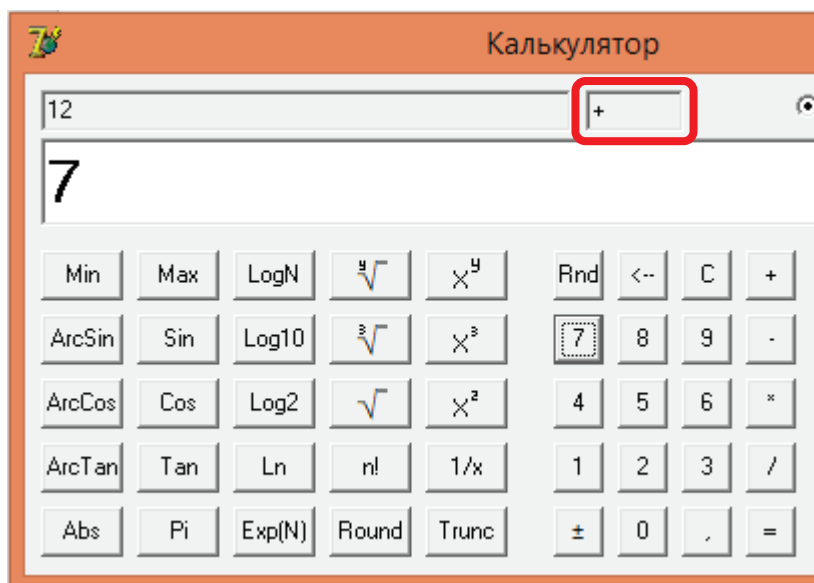


Рисунок 34 – Запомненная операция сложения

Тогда код для кнопки «=» может иметь, например, следующий вид:

```
//Вычислить результат
procedure TForm1.Button20Click(Sender: TObject);
var R1, R2: Real;
    Op: String;
begin
    //Входные значения
    R1 := StrToFloat(Edit1.Text);
    R2 := StrToFloat(Edit2.Text);
    Op := Edit3.Text;

    //Арифметические операции
    if Op = '+' then Edit1.Text := FloatToStr(R2 + R1);
    ...
end;
```

Для вычитания, умножения и деления можно аналогично продолжить код:

```
//Арифметические операции
if Op = '+' then Edit1.Text := FloatToStr(R2 + R1);
if Op = '-' then Edit1.Text := FloatToStr(R2 - R1);
if Op = '*' then Edit1.Text := FloatToStr(R2 * R1);
if Op = '/' then Edit1.Text := FloatToStr(R2 / R1);
...

```

1.16. Цикл for

Цикл **for** – это цикл с фиксированным единичным шагом. Данный вид цикла всегда выполняется конечное количество раз, т.е. его нельзя зациклить.

```
for i := 0 to N-1 do
begin
...
end;
```

Читается как «Для i от 0 до $N-1$ делать», или «Для i от 0 до $N-1$ выполнить».

Где i – счетчик. Всегда целое число (или др. перечислимый тип);

!!! Переменную **i: Integer** необходимо объявить выше в блоке **var**.

0 – начальное значение счетчика;

$N-1$ – конечное значение счетчика;

N – количество шагов.

Счетчик не обязательно начинается с «нуля». Например, счет можно начать с «единицы»:

```
for i := 1 to N do
begin
...
end;
```

Цикл, содержащий единственную команду, может записываться в одну строку (без «**begin**» и «**end**»). Например, вычислим факториал для заданного (заранее) числа N :

```
//Вычисляем факториал N
P := 1;
for i := 1 to N do P := P*i;
```

!!! Этот пример стоит запомнить, т.к. он понадобится нам при доработке «Калькулятора» (раздел 1.9).

Цикл **for** может иметь шаг «**-1**», тогда вместо «**to**» используется «**downto**»:

```
for i := 10 downto 1 do
begin
...
end;
```

2. ОСНОВЫ DELPHI

Embarcadero Delphi (ранее – Borland Delphi и CodeGear Delphi) – интегрированная среда разработки программного обеспечения (ПО) на языке Delphi (ранее носившем название Object Pascal), созданная первоначально фирмой **Borland** и на данный момент принадлежащая и разрабатываемая Embarcadero Technologies.

Создаваемые программы независимы от стороннего ПО, как **Microsoft .NET Framework** или **Java Virtual Machine**. Среда предназначена для быстрой (**RAD**) разработки прикладного ПО для операционных систем **Windows, Linux, Mac OS X**, а также мобильных операционных систем **Android** и **iOS**.

RAD (Rapid Application Development – быстрая разработка приложений) – концепция организации процесса разработки программных продуктов, ориентированная на максимально быстрое получение результата в условиях сильных ограничений по срокам и бюджету и нечетко определенных требований к продукту. Эффект ускорения разработки достигается путем использования соответствующих технических средств и непрерывного, параллельного с ходом разработки, уточнения требований и оценки текущих результатов с привлечением заказчика.

Тот же термин используется в отношении программных инструментов быстрого прототипирования и разработки ПО. Типичными качествами таких инструментов являются максимальная автоматизация рутинных операций и широкое использование визуального программирования.

Delphi был одной из первых **RAD**-систем в мире.

Embarcadero RAD Studio (ранее Borland Developer Studio) – среда быстрой разработки приложений (**RAD**) фирмы Embarcadero Technologies, работающая под Windows. Объединяет **Delphi** и «**C++ Builder**» в единую интегрированную среду разработки. *Таким образом, изучив Delphi, мы сразу освоим и половину C++ Builder-a. При этом Builder позволяет использовать одновременно в одной программе как код написанный на C++, так и модули, написанные на Pascal (что не удивительно, т.к. библиотеки для RAD Studio уже написаны на Delphi).*

IDE (Integrated Development Environment – Интегрированная среда разработки) – комплекс программных средств, используемый программистами для разработки ПО.

Среда разработки включает в себя:

- текстовый редактор;
- транслятор (компилятор и/или интерпретатор);
- средства автоматизации сборки.

Может также включать:

- инструменты для конструирования графического интерфейса пользователя;
- инспектор объектов;
- браузер классов;
- и т.п.

Известное программное обеспечение, созданное на **Delphi**:

- продукция Embarcadero (и ранее Borland): **Delphi**, **C++ Builder**, **JBuilder**. *Важно отметить, что разработчики не всегда пользуются собственным ПО, и многие языки программирования написаны на Си, или каких-либо других языках;*
- инженерное программное обеспечение: **Altium Designer**;
- файловый менеджер: **Total Commander**;
- редактор графики: **IcoFX**;
- аудио-проигрыватель: **AIMP**;
- программы обмена сообщениями: **Skype** (до покупки Microsoft);
- создание музыки: **FL Studio (FruityLoops)**, **Guitar Pro** (до версии 6.0);
- разработка программного обеспечения: **Inno Setup**, **PE Explorer**, **Resource Hacker**;
- веб-разработка: **Macromedia HomeSite**.

2.1. Расширения файлов Delphi

При работе с Delphi используются следующие расширения файлов, представленные в таблице 17.

Таблица 17 – Расширения файлов

Расширение файла	Описание
*.exe	Скомпилированное приложение. Исполняемый файл Windows
*.dll	Динамически подключаемая библиотека Windows
*.pas	Исходный код <u>модуля</u> на языке Паскаль
*.dfm	Исходный код <u>формы</u> (в него автоматически сохраняются все компоненты, размещаемые на форме, их положение и настройки). <i>Файл *.dfm всегда идет в паре с файлом *.pas (с тем же именем!). Эта пара файлов образует оконный модуль. При этом файл *.pas может быть и без файла *.dfm, в этом случае он является обычным («безоконным») модулем</i>
*.dpr	<u>Главный</u> файл проекта Delphi. <i>Исходный код на языке Паскаль. Создается и изменяется автоматически, его не следует редактировать вручную</i>
*.dproj	Главный файл проекта в новых версиях Delphi. <i>При этом старый файл *.dpr также остается. Оба файла будут иметь одно и то же имя, и оба будут главными файлами проекта</i>

Расширение файла	Описание
*.res	<u>Ресурсы</u> проекта. Например, сюда сохраняется изображение с иконкой для будущего exe-файла. Имя res-файла совпадает с соответствующим именем проекта (dpr-файлом)
*.dcu	Скомпилированный модуль. Имеет тоже имя, что и соответствующий ему pas-файл. Является внутренним файлом Delphi. Может быть без проблем удален из папки с проектом (при условии, что имеется pas-файл с таким же именем), т.к. будет пересоздан при следующей компиляции
Файлы с «волной»: *~pas, *~dfm, *~dpr и т.п...	Резервные копии файлов (предыдущее удачное сохранение). Если последнее сохранение прошло успешно, то эти файлы могут быть без проблем удалены из папки с проектом. Если сохранение было неудачным, то необходимо переименовать файл с волной в файл без волны и далее использовать его (т.е. восстановиться из резервной копии)
*.dpr	Главный файл пакета Delphi

2.2. *Версии Delphi

В таблице 18 указаны номера версий Delphi/Pascal и связанные с каждой из них версии компиляторов, начиная с Turbo Pascal 4.0.

Таблица 18 – Версии Delphi

VER<nnn>	Продукт	Версия продукта	Версия пакета	Версия компилятора
VER350	Delphi 11.0 Alexandria / C++Builder 11.0 Alexandria	28	280	35,0
VER340	Delphi 10.4 Sydney / C++Builder 10.4 Sydney	27	270	34,0
VER330	Delphi 10.3 Rio / C++Builder 10.3 Rio	26	260	33,0
VER320	Delphi 10.2 Tokyo / C++Builder 10.2 Tokyo	25	250	32,0
VER310	Delphi 10.1 Berlin / C++Builder 10.1 Berlin	24	240	31,0
VER300	Delphi 10 Seattle / C++Builder 10 Seattle	23	230	30,0
VER290	Delphi XE8 / C++Builder XE8	22	220	29,0
VER280	Delphi XE7 / C++Builder XE7	21	210	28,0
VER270	Delphi XE6 / C++Builder XE6	20	200	27,0

VER<nnn>	Продукт	Версия продукта	Версия пакета	Версия компилятора
VER260	Delphi XE5 / C++Builder XE5	19	190	26,0
VER250	Delphi XE4 / C++Builder XE4	18	180	25,0
VER240	Delphi XE3 / C++Builder XE3	17	170	24,0
VER230	Delphi XE2 / C++Builder XE2	16	160	23,0
VER220	Delphi XE / C++Builder XE	15	150	22,0
VER210	Delphi 2010 / C++Builder 2010	14	140	21,0
VER200	Delphi 2009 / C++Builder 2009	12	120	20,0
VER190	Delphi 2007 for .Net	11	110	19,0
VER180 <i>u</i> VER185	Delphi 2007 / C++Builder 2007 for Win32	11	110	18,5
VER180	Delphi 2006 / C++Builder 2006	10	100	18,0
VER170	Delphi 2005	9	90	17,0
VER160	Delphi 8 for .Net	8	80	16,0
VER150	Delphi 7 (и 7.1)	7	70	15,0
VER140	Delphi 6 / C++Builder 6	6	60	14,0
VER130	Delphi 5 / C++Builder 5	5	NA	NA
VER125	C++Builder 4	4	NA	NA
VER120	Delphi 4	4	NA	NA
VER110	C++Builder 3	3	NA	NA
VER100	Delphi 3	3	NA	NA
VER93	C++Builder 1	NA	NA	NA
VER90	Delphi 2	2	NA	NA
VER80	Delphi 1	1	NA	NA
VER70	Borland Pascal 7.0	NA	NA	NA
VER15	Turbo Pascal for Windows 1.5	NA	NA	NA
VER10	Turbo Pascal for Windows 1.0	NA	NA	NA
VER60	Turbo Pascal 6.0	NA	NA	NA
VER55	Turbo Pascal 5.5	NA	NA	NA
VER50	Turbo Pascal 5.0	NA	NA	NA
VER40	Turbo Pascal 4.0	NA	NA	NA

2.3. Библиотека VCL

VCL (Visual Component Library – Библиотека визуальных компонентов) – объектно-ориентированная библиотека для разработки программного обеспечения, созданная компанией **Borland**. Входит в комплект поставки **Delphi**, **C++ Builder** и **RAD Studio**, и является частью среды разработки. *Все компоненты, которые мы ранее добавляли на форму (да и сама форма), это все была VCL.* Библиотека написана на Паскаль.

Компоненты добавляются из «**Палитры компонентов**». В старых версиях (таких как **Delphi 7**) она расположена сверху (рис. 35), в новых версиях (например, **Delphi XE8**) она расположена справа снизу (рис. 36).

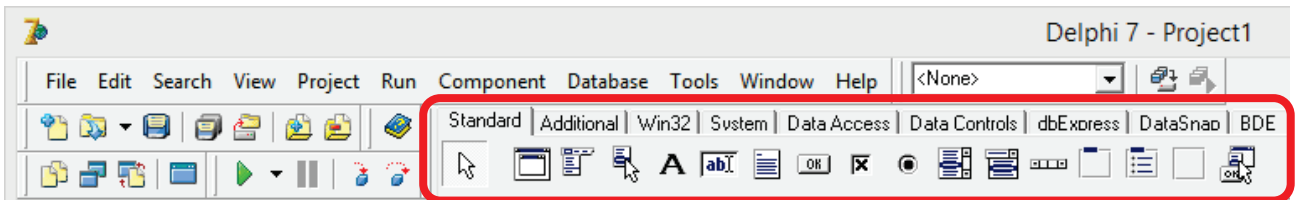


Рисунок 35 – Палитра компонентов Delphi 7

Все компоненты в Палитре являются наследниками класса **TComponent**. При этом все компоненты делятся на две группы: «**Визуальные компоненты**» и «**Невизуальные компоненты**».

«**Визуальные компоненты**» (или «**Контролы**») – элементы пользовательского интерфейса. Относятся к классу **TControl**, который, в свою очередь, сам является компонентом (наследником класса **TComponent**). Примерами визуальных компонентов являются: **Button**, **Edit**, **Label**, **Memo**, **CheckBox**, **Listbox** и многие др. Они всегда видны на экране и выглядят одинаково на стадии проектирования и во время работы приложения.

Сама Форма (т.е. окно будущего приложения), на которую добавляются все остальные компоненты, также является **Контролом**.

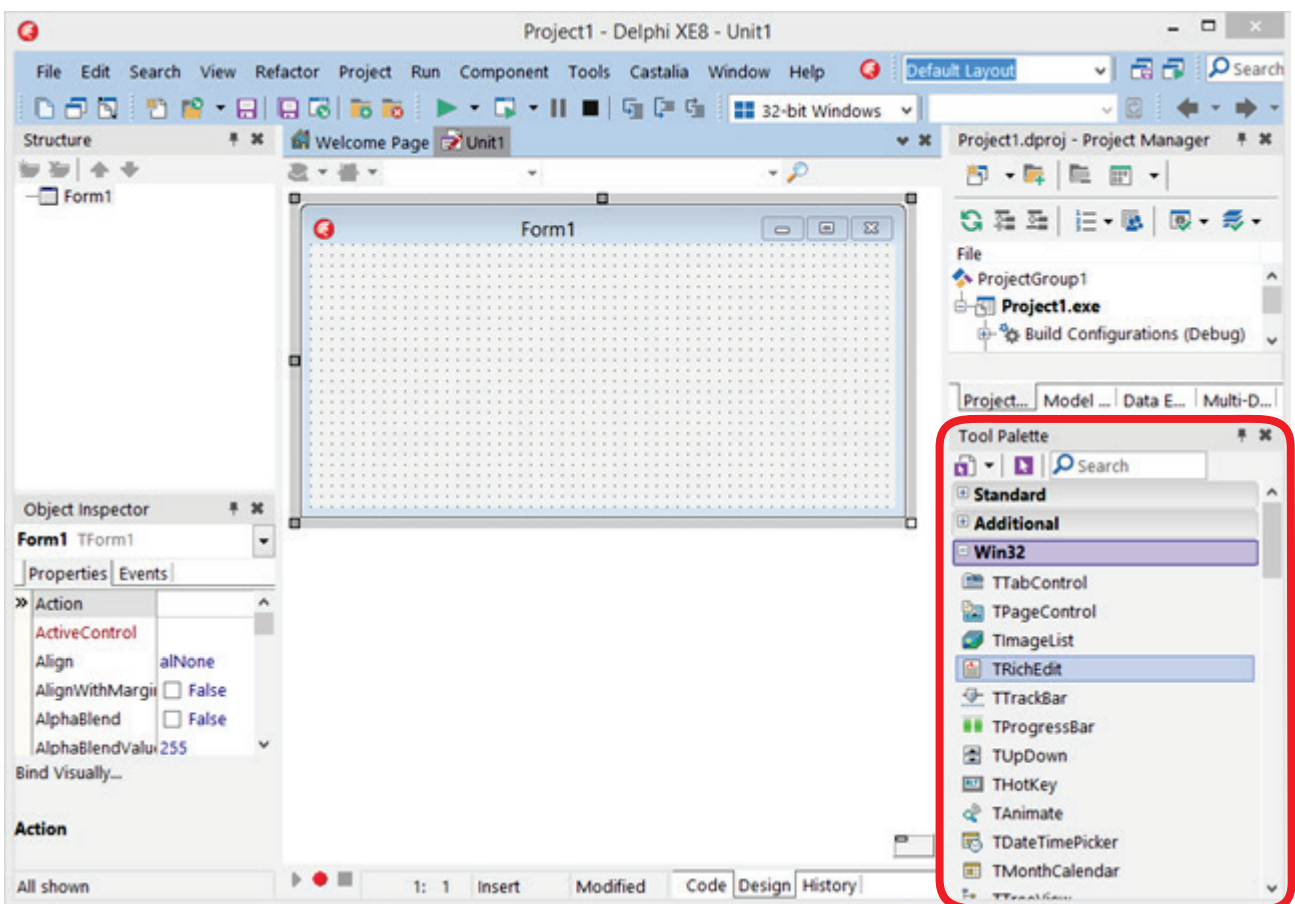


Рисунок 36 – Палитра компонентов Delphi XE8

«**Невизуальные компоненты**» – это все компоненты (**TComponent**), кроме Визуальных (**TControl**). Для них характерно то, что на Форме они всегда выглядят в виде небольших квадратиков с изображением, их размер

невозможно изменить (хотя их и можно перемещать по Форме). Эти «квадратики» видны только на этапе проектирования, но они не будут видны после запуска приложения, поэтому они и называются «Невизуальными».

Примерами невидимых компонентов являются: **Timer**, **MainMenu**, **PopupMenu**, **ImageList**, **ActionList** и многие др.

Design-time (этап проектирования, этап разработки) – один из этапов жизненного цикла приложения, на котором оно создается программистом. Прежде всего здесь имеется в виду работа с графической частью программы (размещение компонентов на форме, выбор их положения и свойств, связывание компонентов и т.п.).

Run-time (этап выполнения, время исполнения и т.п.) – один из этапов жизненного цикла приложения, после его запуска (нажатия **F9**). Это основной этап, ради которого и создается приложение. На этом этапе для приложения уже создан **exe-файл**. *Обычно к **Run-time** мы будем относить и Код программы, который мы набираем в виде текста. Хотя сам код и пишется на этапе проектирования, но работает он только после запуска программы (**F9**).*

Существуют и другие этапы жизненного цикла приложения, например, после нажатия **F9**, но до старта приложения, выполняется этап компиляции (**Compile-time**). Если на этапе компиляции обнаружатся ошибки, то этап исполнения (**Run-time**) не наступит вовсе.

Все компоненты, добавляемые на Форму в **Design-time**, отображаются в «Дереве объектов» («**Object TreeView**»), расположенном слева наверху (рис. 37). Если выделить компонент на Форме, то он выделится и в Дереве (и наоборот).

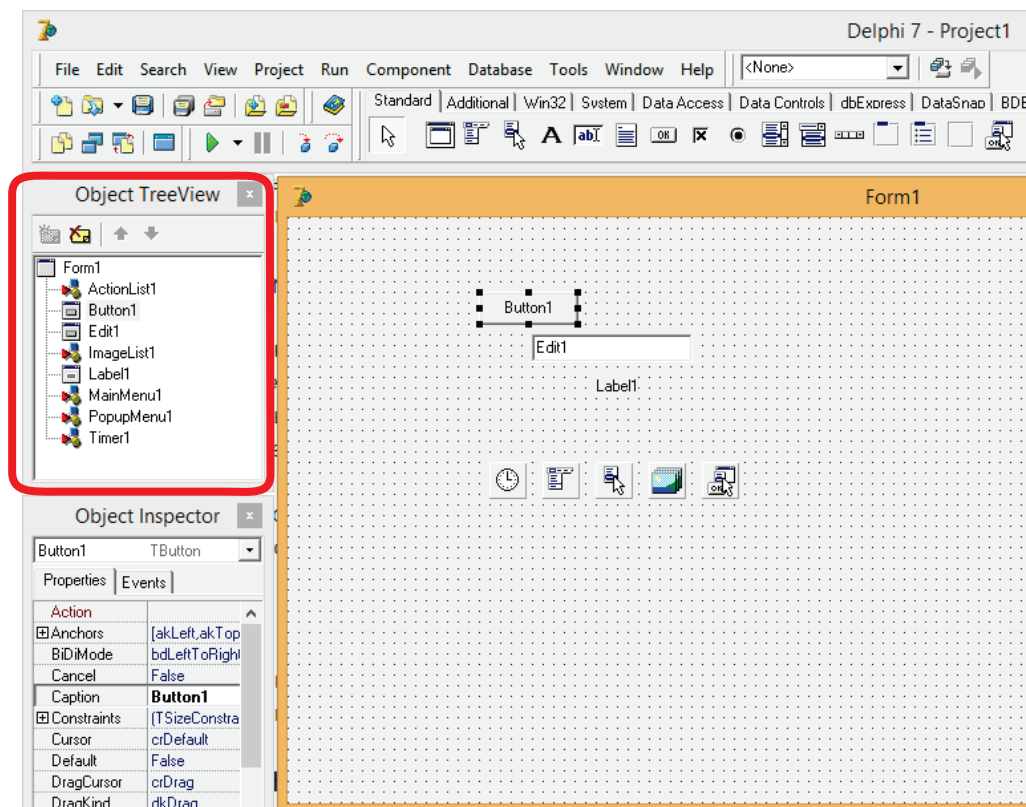


Рисунок 37 – Дерево объектов (Object TreeView)

Некоторые из компонентов в Дереве объектов могут быть вложены (рис. 38) в другие компоненты.

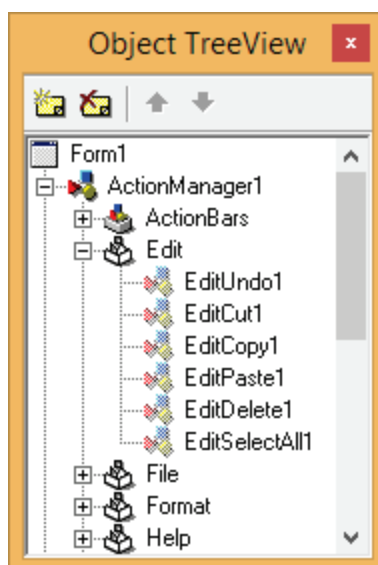


Рисунок 38 – Вложенные объекты

2.4. Свойства визуальных компонентов

«Инспектор объектов» («Object Inspector») расположен слева внизу (рис. 39), с его помощью осуществляется настройка Свойств Компонентов (Объектов). Если выбрать Компонент на Форме или в Дереве объектов, то он отобразится и в Инспекторе объектов.

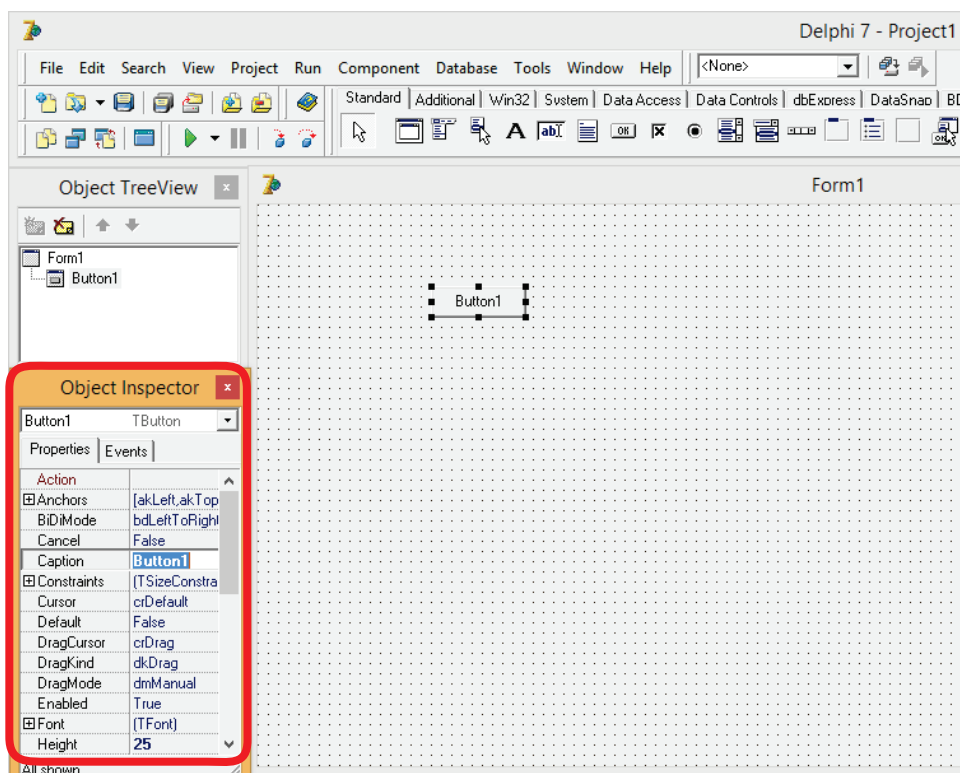


Рисунок 39 – Инспектор объектов

Все добавляемые компоненты уже имеют настроенные по умолчанию свойства. Если изменить свойство компонента, то его значение будет выделено **жирным** (рис. 40). Если изменить его обратно на значение по умолчанию, то оно снова перестанет быть жирным. *Некоторые свойства всегда выделены жирным (т.к. для них разработчиками не предусмотрены значения по умолчанию).*

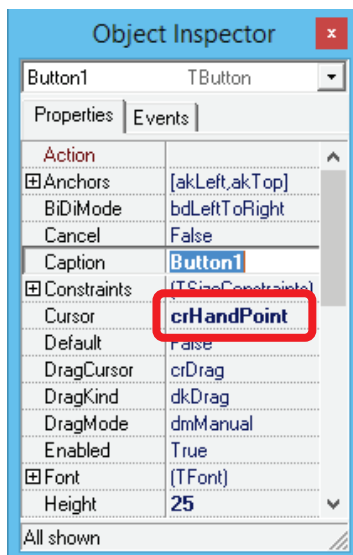


Рисунок 40 – Свойство, выделенное жирным

Свойства Компонентов можно изменять как в **Design-time** – через Инспектор объектов (рис. 41), так и в **Run-time** – т.е. через код программы. Например:

```

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

```

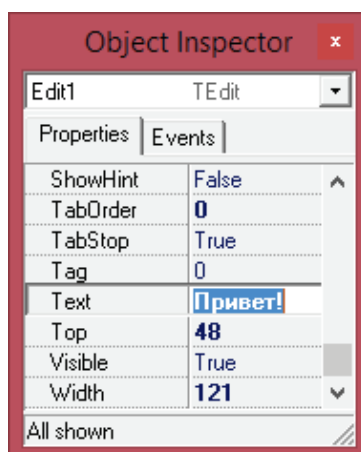


Рисунок 41 – Настройка свойства Text

Свойства визуальных компонентов

Основные Свойства визуальных компонентов представлены в таблице 19. В отдельную таблицу вынесены Свойства, задающие надписи и изображения на различных компонентах (табл. 20).

!!! На данном этапе лучше запустить *Delphi* и протестировать свойства для указанных компонентов. Сейчас нас интересует их настройка в *Design-time* через Инспектор объектов, а не написание их в коде!

Таблица 19 – Основные свойства визуальных компонентов

Свойство	Описание
Name: String;	Имя компонента. Это Имя, отображаемое в Дереве объектов, и то Имя, по которому можно обратиться к компоненту из кода программы. !!! Имя <u>не</u> должно содержать пробелов и русских букв! Имя может состоять только из английских букв, цифр и символа подчеркивания. Причем имя не может начинаться с цифры. <i>По умолчанию для нового компонента задается имя, полученное из его типа, с прибавлением порядкового номера, например, для TButton получаем: Button1, Button2, Button3 и т.д.</i>
Visible: Boolean;	Видимость компонента. Visible := False – делает компонент невидимым (скрытым)
Enabled: Boolean;	Доступность компонента. Определяет, реагирует ли компонент на нажатия, мышки и клавиатуры. <i>Для большинства компонентов при Enabled := False, текст и изображение на компоненте становятся <u>серыми</u>, подчеркивая что он недоступен</i>
Left: Integer; Top: Integer;	Координаты левого верхнего края компонента в пикселях. <i>В Delphi (как и вообще в Windows) координата у отсчитывается <u>сверху вниз</u> (а не как принято в математике, снизу вверх)</i>
Width: Integer; Height: Integer;	Ширина и высота. Горизонтальный и вертикальный размеры компонента в пикселях
Align: TAlign;	Определяет способ выравнивания компонента в контейнере (в родительском компоненте, на форме, в окне). Align := alNone – нет выравнивания. Align := alClient – занять все возможное пространство (всю клиентскую область). alTop, alBottom, alLeft, alRight – выравнивание по верхнему, нижнему, левому или правому краю

Свойство	Описание
Color: TColor;	Цвет фона компонента. Например: Color := clBtnFace;
Font: TFont;	Шрифт текста. Включает в себя такие свойства как: Font. Size – размер шрифта; Font. Color – цвет текста; Font. Name – имя шрифта; Font. Style – позволяет сделать текст жирным, курсивным, подчеркнутым или зачеркнутым.
Cursor: TCursor;	Определяет вид курсора мышки, когда тот находится на данном компоненте. Например: Cursor := crArrow – обычная стрелка курсора; Cursor := crHandPoint – курсор в виде руки с пальцем; Cursor := crCross – курсор в виде перекрестия (<i>для рисования</i>); и др.
ReadOnly: Boolean;	«Только для чтения». Запрещает ввод значений в данное поле. <i>Применяется только для тех компонентов, в которые можно ввести какой-то текст, или числовое значение, например: Edit, Memo, SpinEdit</i>
WordWrap: Boolean;	«Переносить по словам». Разбивает текст на несколько подстрок, если он не влезает в одну строку. Разделение происходит по пробелам (т.е. целыми словами)
AutoSize: Boolean;	Автоматически подгонять размеры компонента под его содержимое (как правило, под размер текста)
Hint: String; ShowHint: Boolean;	Текст всплывающей подсказки и разрешение отображать эту подсказку. Подсказка появляется, когда курсор мышки подносится к соответствующему компоненту. По умолчанию подсказка появляется на желтом фоне. <i>Обычно ShowHint задается не для конкретного компонента, а для Формы, что разрешает (или запрещает) сразу все всплывающие подсказки в этом окне, например:</i> <i>Form1.ShowHint := True</i>
ParentColor: Boolean; ParentFont: Boolean; ParentShowHint: Boolean;	Использовать родительские настройки цвета, шрифта и доступности всплывающей подсказки. <i>Их лучше не менять без особой необходимости</i>

Свойство	Описание
TabOrder: -1..32767; TabStop: Boolean;	Порядок обхода компонентов при нажатии клавиши Tab и разрешение компоненту участвовать в таком обходе. <i>Имеет смысл только для компонентов, в которые можно что-то ввести (например, TEdit), или что-то нажать (например, TButton). Эти свойства не существуют, например, у TLabel, или TImage</i>

Таблица 20 – Свойства, задающие надписи и изображения

Свойство	Описание
Caption: String; <i>или</i> Text: String;	Текстовая надпись на компоненте или заголовок Формы (заголовок окна)
Lines: TStrings; <i>или</i> Items: TStrings; <i>или</i> Strings: TStrings;	Многострочный текст (Memo, RichEdit) или пункты списков (ListBox, ComboBox, RadioGroup, CheckListBox, ValueListEditor)
Value: Integer;	Числовое значение, для полей ввода чисел (SpinEdit)
Checked: Boolean; <i>или</i> Down: Boolean;	Логические (Булевы) значения. Установленные «галочки» (CheckBox, RadioButton) или зажатые кнопки, допускающие «залипание» в нижнем положении (ToolButton, SpeedButton)
Picture: TPicture; <i>или</i> Glyph: TBitmap; <i>или</i> ImageIndex: Integer;	Изображение. Для Picture и Glyph изображение загружается непосредственно в них. Для ImageIndex выбирается только номер изображения, а само изображение должно быть заранее загружено в список изображений ImageList

Свойства-привязки

Некоторые Свойства позволяют «привязываться» к другим компонентам, имена таких Свойств будут выделены в Инспекторе объектов красным (рис. 42). Их можно выбрать из выпадающего списка (*если список пуст, то необходимо вначале добавить на Форму соответствующие компоненты для привязки*). Основные привязки компонентов перечислены в табл. 21.

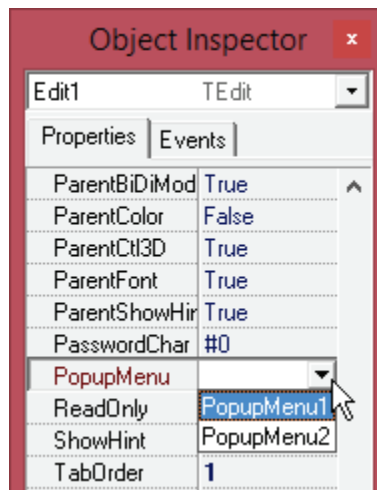


Рисунок 42 – Привязка всплывающего меню

Таблица 21 – Основные привязки компонентов

Свойство	Описание
Images: TImageList;	Привязка списка изображений
PopupMenu: TPopupMenu;	Привязка всплывающего меню
Action: TBasicAction;	Действие (Action), связанное с данным компонентом (кнопкой или пунктом меню)

Свойства формы

Форма значительно отличается от других визуальных компонентов, т.к. ей приходится взаимодействовать не только с объектами программы, но и с внешним миром. Поэтому у нее имеется собственный набор свойств (табл. 22). Большинство из этих свойств «придумано» не разработчиками **Delphi**, а произрастает из операционной системы **Windows**, которая сама отвечает за взаимодействие окон и за их отображение.

Таблица 22 – Основные свойства формы

Свойство	Описание
Caption: String;	Заголовок окна
Icon: TIcon;	Иконка в заголовке окна (верхний левый угол). <i>Файл с иконкой (в формате *.ico) необходимо подготовить заранее</i>
Menu: TMainMenu;	Сюда привязывается главное меню (MainMenu). <i>Имеет смысл изменять только в том случае, когда на форме имеется несколько компонентов TMainMenu, между которыми нужно переключаться, или когда необходимо включать и отключать главное меню</i>
AutoSize: Boolean;	Автоматически подгоняет размер окна, под содержащиеся в нем компоненты
AutoScroll: Boolean;	Отображает полосы прокрутки, в случае, если не все компоненты помещаются на форму

Свойство	Описание
Width: Integer; Height: Integer;	Ширина и высота окна. Горизонтальный и вертикальный размеры окна в пикселях
BorderWidth: Integer; ClientWidth: Integer; ClientHeight: Integer;	Ширина границы внутри окна, а также ширина и высота клиентской области (т.е. той области, где мы можем размещать визуальные компоненты). <i>Клиентская область не может быть больше размеров окна (ширины Width и высоты Height окна). Обычно она меньше, т.к. у окна есть рамки, заголовок, главное меню и т.п.</i> <i>BorderWidth также сужает клиентскую область</i>
FormStyle: TFormStyle;	Стиль отображения окна. FormStyle := fsNormal – обычное отображение окна; FormStyle := fsStayOnTop – «поверх других окон» <i>(окно остается видимым, даже если активно окно другой программы)</i>
Position: TPosition;	Начальное положение окна при старте программы. Position := poDesigned – то положение, в котором окно было при разработке в Design-time; Position := poScreenCenter – по центру экрана; Position := poMainFormCenter – поместить окно по центру главного окна приложения (имеет смысл только для дочерних окон); и др.
WindowState: TWindowState;	Состояние окна. WindowState := wsNormal – обычное состояние; WindowState := wsMaximized – развернуто на весь экран; WindowState := wsMinimized – свернуто
BorderStyle: TFormBorderStyle;	Определяет тип заголовка окна и рамки вокруг окна. BorderStyle := bsSizeable – обычное состояние (размеры окна можно менять, есть заголовок и кнопки в заголовке); BorderStyle := bsSingle – тоже-самое, но размеры окна нельзя изменять (но при этом окно все еще можно развернуть на весь экран); BorderStyle := bsNone – полное отсутствие заголовка окна и возможности изменять размеры окна или перемещать его по экрану <i>(такое окно можно закрыть нажатием Alt+F4)</i> ;

Свойство	Описание
	<p>BorderStyle := bsDialog – упрощенный заголовок, содержащий единственную кнопку «Заккрыть» (нет кнопок «Свернуть» и «Развернуть»), размеры окна не изменяются. Применяется для диалоговых окон;</p> <p>BorderStyle := bsToolWindow – аналогичен предыдущему (отличается внешним видом кнопки «Заккрыть»);</p> <p>BorderStyle := bsSizeToolWin – единственная кнопка «Заккрыть», но можно изменять размеры окна.</p>
<p>BorderIcons: TBorderIcons;</p>	<p>Составное свойство, позволяющее сделать недоступными кнопки заголовка.</p> <p>Включает в себя следующие значения типа Boolean:</p> <p>biMinimize – запретить кнопку сворачивания окна;</p> <p>biMaximize – запретить кнопку разворачивания окна на весь экран;</p> <p>и др.</p>
<p>Constraints: TSizeConstraints;</p>	<p>Составное свойство, позволяющее ограничить минимальные и максимальные возможные размеры окна.</p> <p>Состоит из:</p> <p>Constraints.MinWidth: Integer;</p> <p>Constraints.MaxWidth: Integer;</p> <p>Constraints.MinHeight: Integer;</p> <p>Constraints.MaxHeight: Integer;</p>
<p>AlphaBlend: Boolean; AlphaBlendValue: Byte;</p>	<p>Делает форму прозрачной.</p> <p>Значение задается от 0 до 255, где 0 – полностью невидимый, а 255 – полностью видимый.</p> <p><i>Может использоваться для анимации с плавным появлением или исчезновением окна</i></p>
<p>TransparentColor: Boolean; TransparentColorValue: TColor;</p>	<p>Задает один из цветов как прозрачный.</p> <p><i>Никак не связано с предыдущим AlphaBlend.</i></p> <p><i>Может применяться для создания окон не только прямоугольной, но и любой произвольной формы</i></p>

2.5. Изменение иконки приложения

Свойство **Icon** формы позволяет задать иконку для окна, но она не изменится у **exe**-файла. Чтобы сменить иконку для всего приложения, необходимо выбрать меню «Project» → «Options» (Ctrl+Shift+F11), и далее на вкладке «Application» нажать кнопку «Load Icon» (рис. 43).

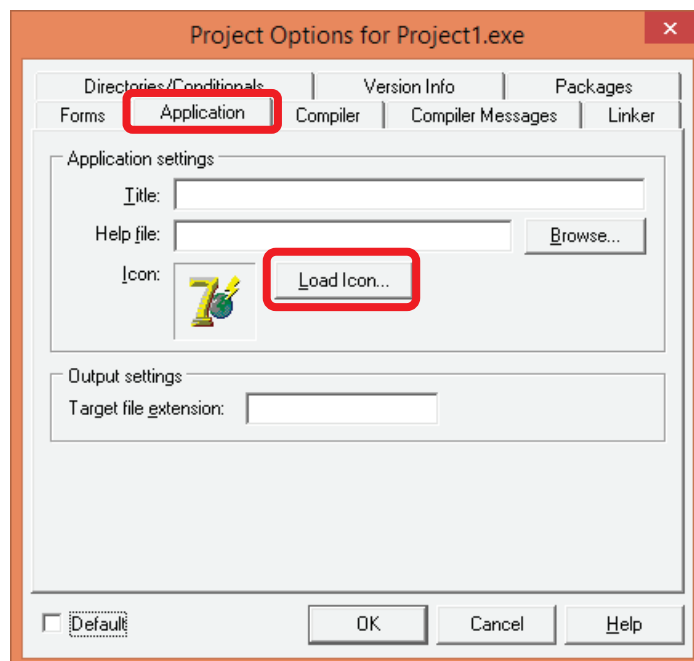






Рисунок 43 – Выбор иконки для приложения

















Файл с иконкой (в формате ***.ico**) необходимо подготовить заранее. Стандартная иконка должна быть **32x32** пикселя.

2.6. Невизуальные компоненты Delphi

Рассмотрим основные невидимые компоненты **Delphi** (табл. 23).

Таблица 23 – Основные невидимые компоненты

Компоненты	Описание
Вкладка «Standard»	
 MainMenu	Главное меню и всплывающее меню
 PopupMenu	
 ActionList	Список Действий (Action-ов)
Вкладка «Additional»	
 ApplicationEvents	События Приложения

Компоненты	Описание
 TrayIcon	Позволяет поместить иконку в Трей (у часов). <i>Отсутствует в Delphi 7</i>
 ActionManager	Менеджер Действий (Action-ов). <i>Расширенный вариант ActionList</i>
 CustomizeDlg	Специальный диалог настройки панелей управления и меню. <i>Применяется только совместно с ActionManager</i>
Вкладка «Win32»	
 ImageList	Список изображений для кнопок и пунктов меню
Вкладка «System»	
 Timer	Таймер
Вкладка «Dialogs»	
 OpenDialog  SaveDialog	Диалоговые окна открытия и сохранения файла
 OpenPictureDialog,  SavePictureDialog	Диалоговые окна открытия и сохранения изображения
 ColorDialog  FontDialog	Диалоговые окна выбора цвета и шрифта
 PrintDialog  PrinterSetupDialog  PageSetupDialog	Диалоговые окна печати, настройки принтера и настройки страницы
 FindDialog  ReplaceDialog	Диалоговые окна поиска и замены текста

2.7. Диалоговые окна

Для запуска любого из диалогов (в **Run-time**), необходимо выполнить для него команду **Execute** («Выполнить»). Например:

```
if OpenDialog1.Execute then
begin
  //Этот код выполняется после нажатия ОК в диалоге
end;
```

!!! Если используются **Action**-ы (см. разделы 2.8 и 2.9), то код для запуска диалога не нужно писать вовсе, т.к. соответствующие **Action**-ы сами умеют вызывать диалоги.

Для просмотра внешнего вида диалога в **Design-time**, достаточно дважды кликнуть по соответствующему невидимому компоненту на форме.

Внешний вид диалогового окна Открытия файла представлен на рисунке 44. Диалог Сохранения файла отличается от него только надписью в заголовке.

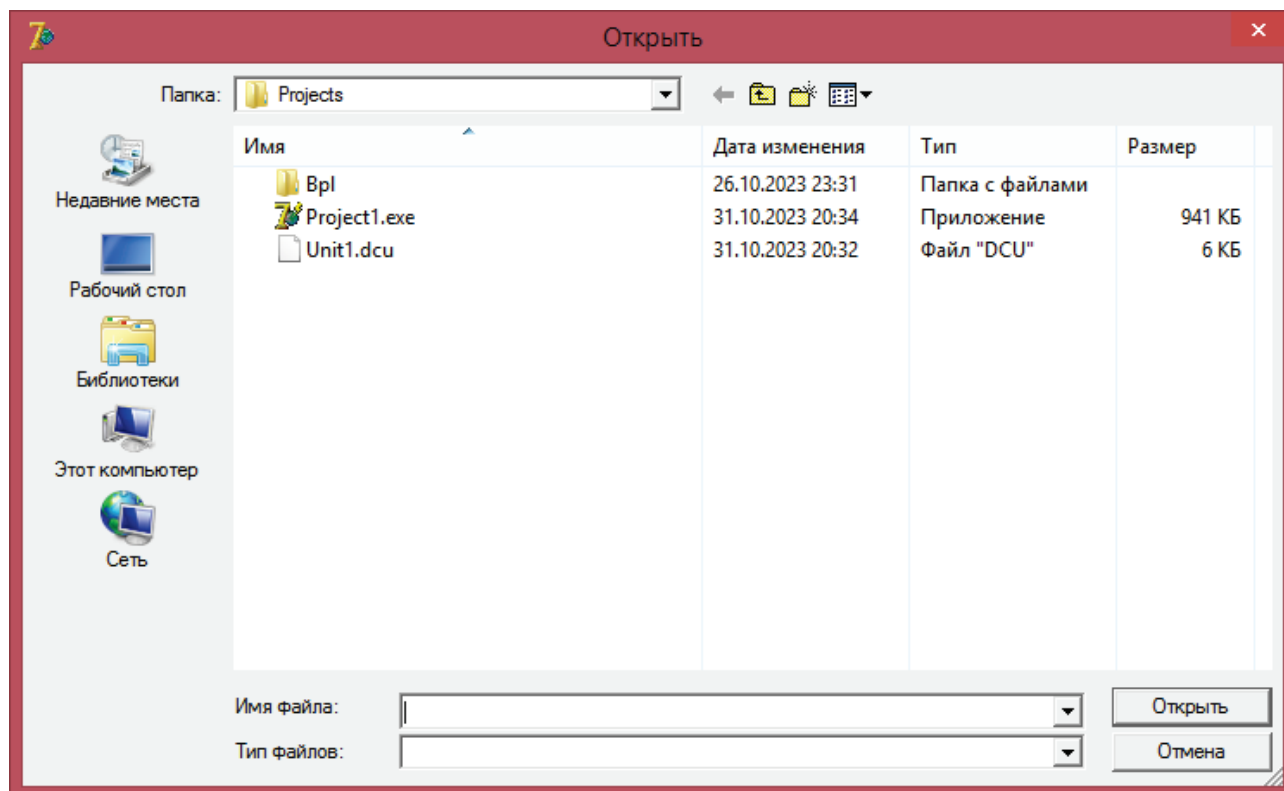


Рисунок 44 – Диалог открытия (сохранения) файла

Диалоги открытия и сохранения Изображения (рис. 45) практически аналогичны диалогам открытия и сохранения файла, но имеют справа область для предварительного просмотра изображения.

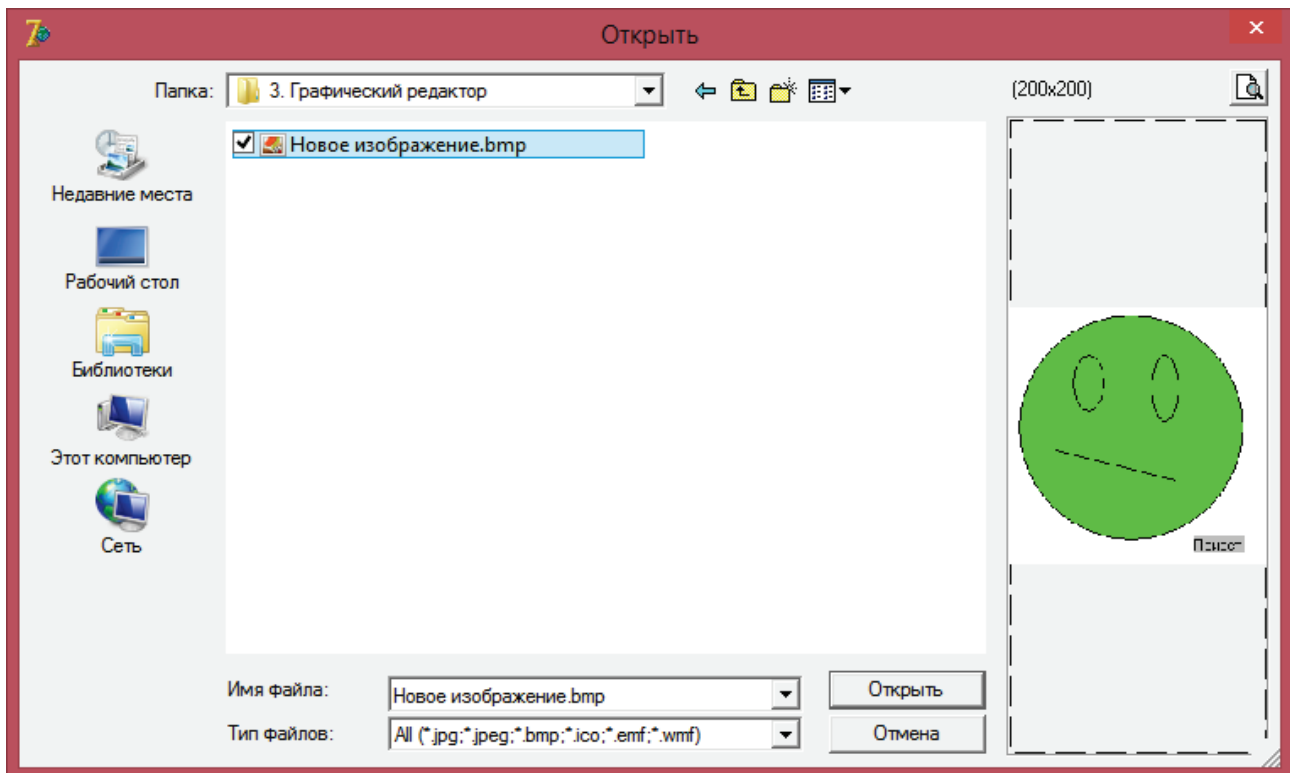


Рисунок 45 – Диалог открытия (сохранения) изображения

Диалоги выбора Цвета и Шрифта представлены на рисунке 46 и рисунке 47.

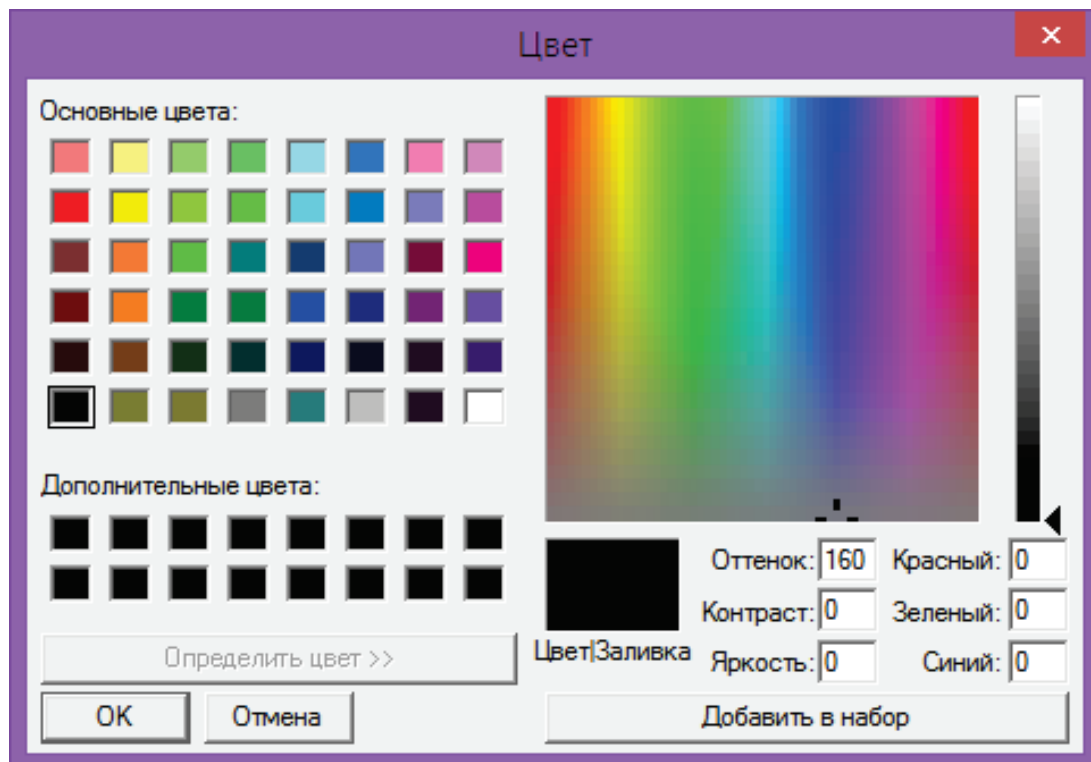


Рисунок 46 – Диалог выбора цвета

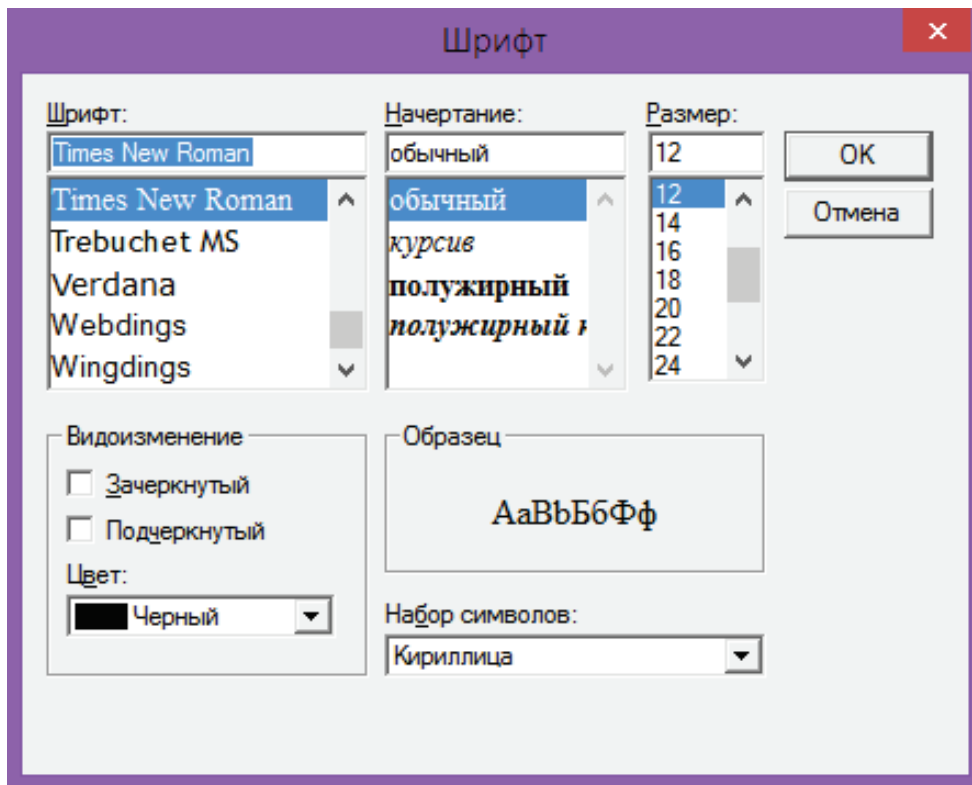


Рисунок 47 – Диалог выбора шрифта

Диалоги Печати и Настройки печати представлены на рисунке 48 и рисунке 49.

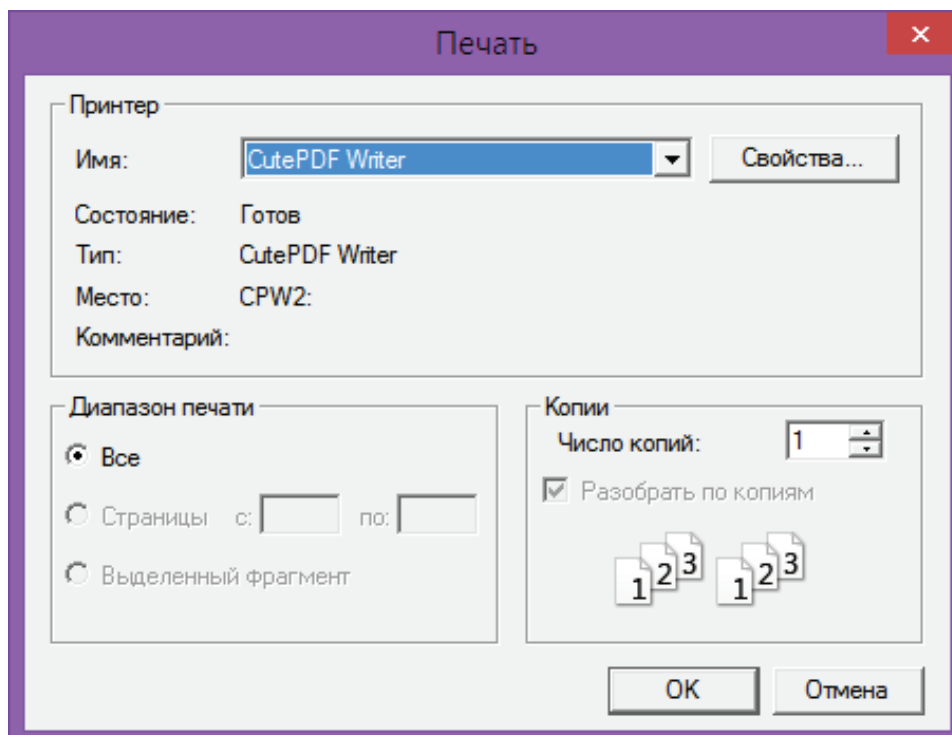


Рисунок 48 – Диалог печати

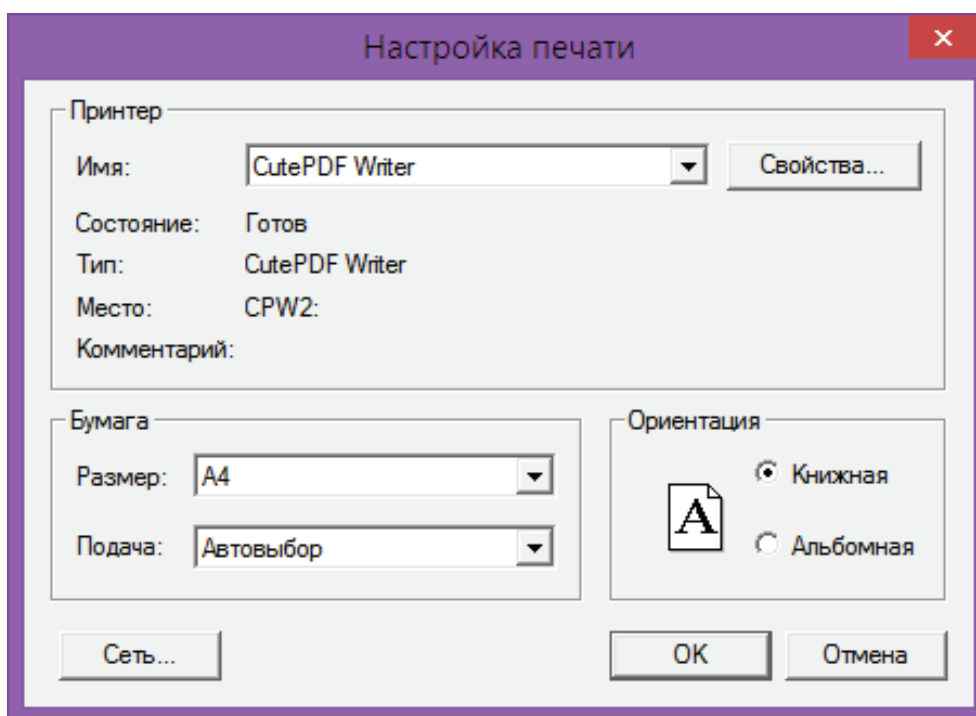


Рисунок 49 – Диалог настройки печати

Диалог Настройки страницы представлен на рисунке 50.

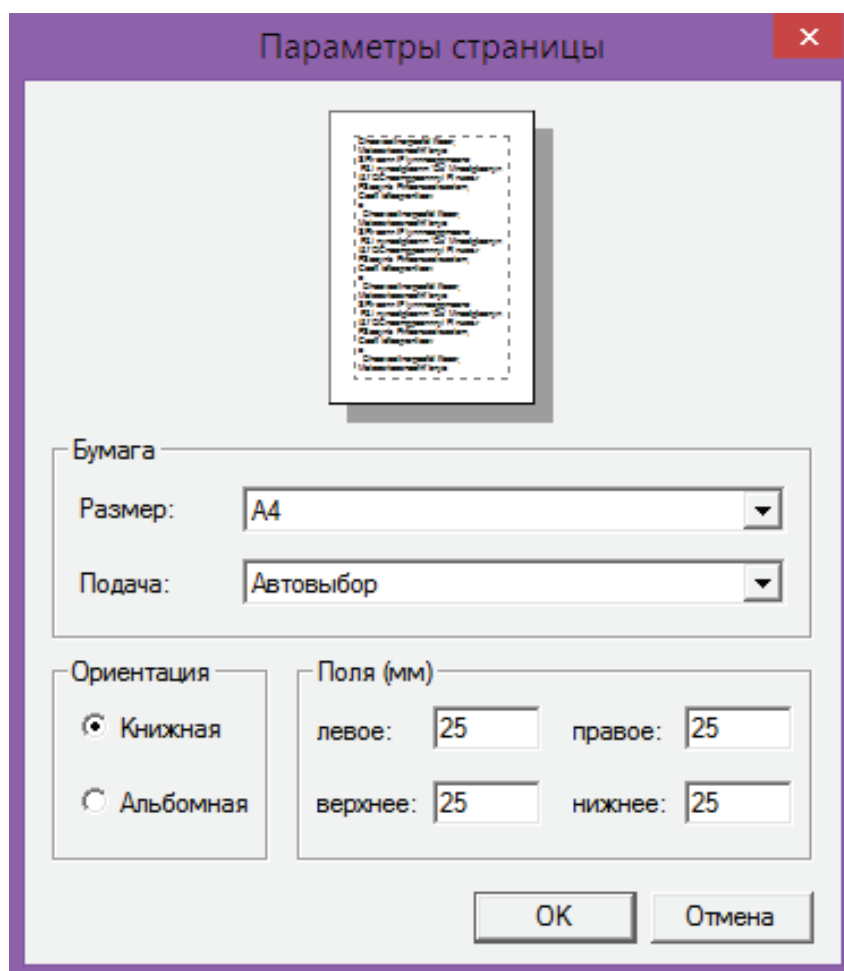


Рисунок 50 – Диалог настройки страницы

Диалоги Поиска и Замены текста представлены на рисунке 51 и рис. 52.

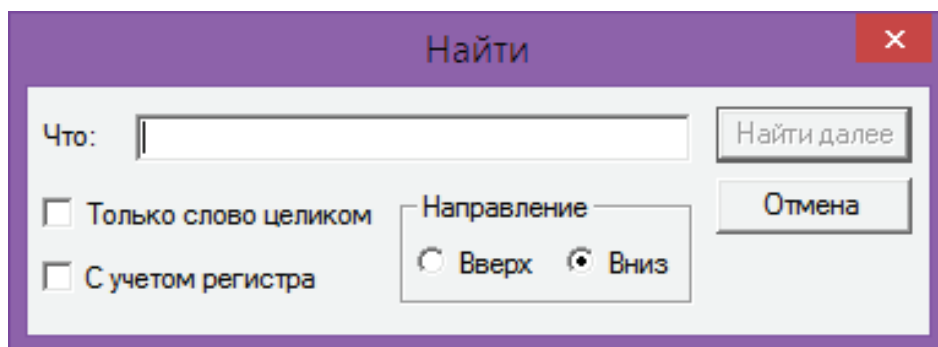


Рисунок 51 – Диалог поиска текста

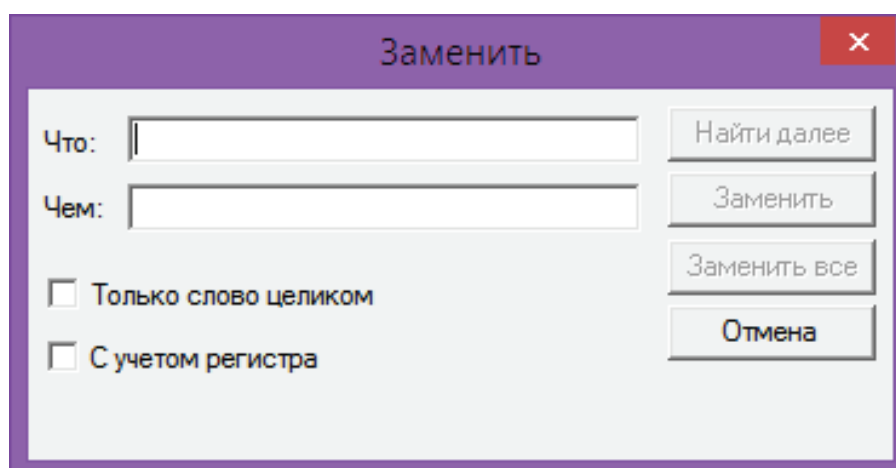


Рисунок 52 – Диалог замены текста

2.8. Механизм Actions

Action (Действие) – реализация некоторого поведения, являющегося реакцией на поступок пользователя, такой, как щелчок на кнопке или выбор пункта меню. В Delphi имеется много стандартных действий для типовых ситуаций (таких как, «Копировать», «Вставить», «Сохранить как», «Печатать» и т.п.).

ActionList – список действий, позволяющий упорядочить работу с Действиями.

ActionManager – менеджер действий – расширенный вариант ActionList.

ActionList (или **ActionManager**) имеет смысл применять в следующих случаях:

- когда одну и ту же команду можно выполнить из разных мест (из главного меню, панели управления или нескольких панелей управления, всплывающего меню и т.п.);
- когда необходимо задать горячие клавиши (**ShortCut**);
- когда необходимо, чтобы пункты меню или кнопки панели управления могли становиться недоступными при определенном условии;

- для крупного проекта, в котором необходимо иметь единый список действий и механизм для управления ими (в том числе иметь возможность централизованно сменить названия, иконки, всплывающие подсказки, горячие клавиши);

- кроме того, применение данного механизма оправдано, когда мы хотим использовать стандартные **Action**-ы, которые уже есть в **Delphi**.

Для **ActionManager** также актуальными могут быть случаи:

- когда разрабатывается проект с несколькими панелями, состав которых планируется неоднократно менять в процессе разработки;

- когда необходимо предоставить пользователю возможность менять порядок кнопок на панели управления прямо во время работы программы, добавлять или удалять кнопки, либо изменять структуру главного меню программы.

Для добавления Действий в проект, необходимо дважды кликнуть по **ActionList**, после чего выбрать «**New Action**» или «**New Standard Action**» (рис. 53).

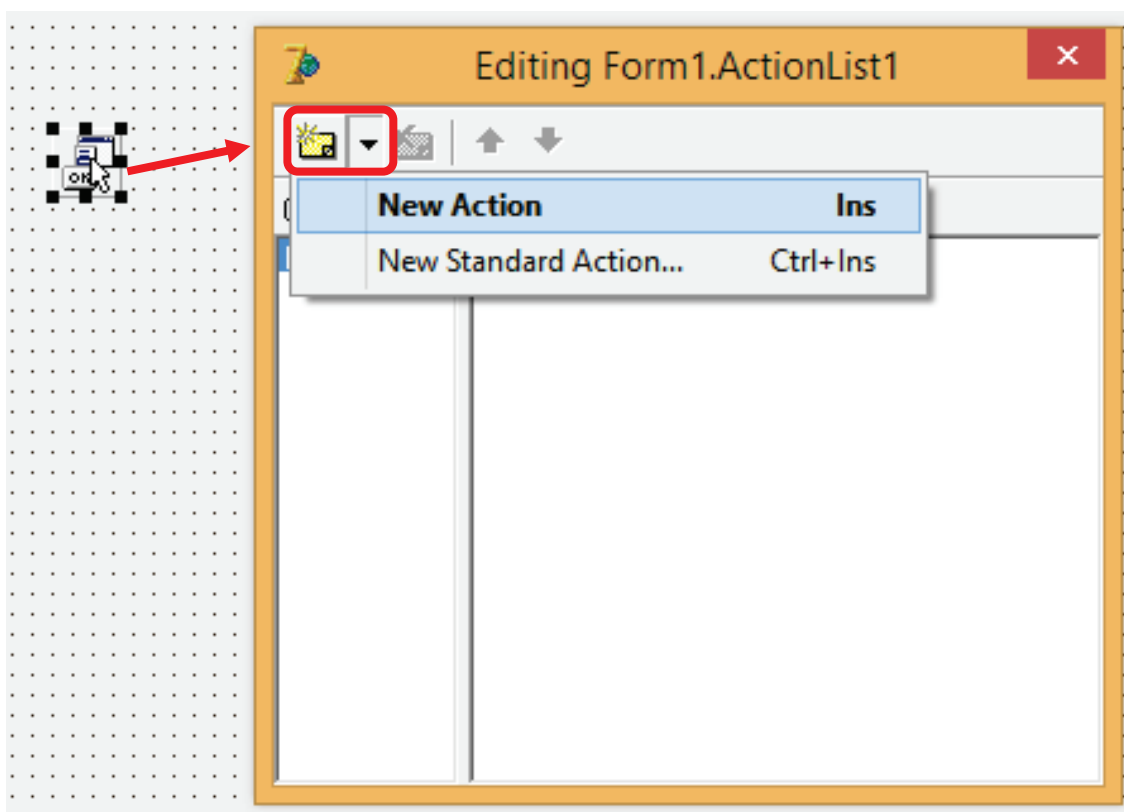


Рисунок 53 – Добавление действий

Каждое Действие **Action** содержит свойства, представленные на рисунке 54 и в таблице 24.

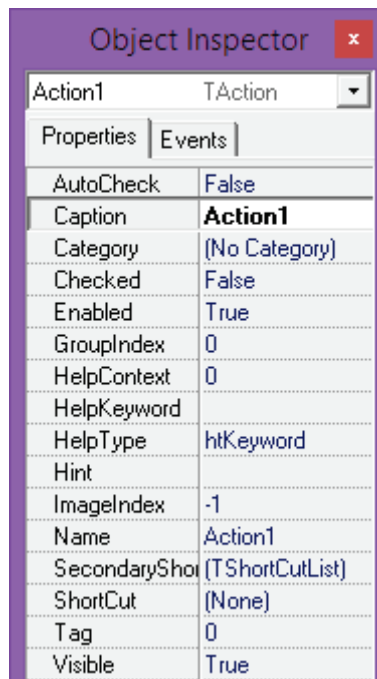


Рисунок 54 – Свойства Action-ов

Таблица 24 – Свойства Action-ов

Свойство	Описание
Name: String;	Название Action-а. Может состоять только из латинских букв, цифр и символа подчеркивания. Для всех собственных Action-ов необходимо обязательно указать <u>осмысленные</u> имена (т.е. не нужно оставлять имена Action1, Action2 и т.д.). <i>Стандартные Action-ы можно не переименовывать</i>
Caption: String;	Надпись, которая будет отображена на кнопке или пункте меню. <i>Пишется по-русски</i>
Category: String;	«Категория» – группа, в которую помещается действие. В Инспекторе объектов можно выбрать одну из уже существующих категорий из <u>выпадающего списка</u> , либо написать любое новое имя категории. Категории не видны конечному пользователю и используется только на этапе проектирования для удобства самого разработчика. <i>Но при использовании ActionManager, категории могут быть еще и разделами меню, и будут видны конечному пользователю</i>
Hint: String;	Текст всплывающей подсказки. Подсказка появляется, когда курсор мышки подносится к соответствующему компоненту. По умолчанию подсказка появляется на желтом фоне. <i>Как и прежде, для отображения подсказок их вначале необходимо включить через ShowHint: Form1.ShowHint := True</i>

Свойство	Описание
Shortcut: TShortcut;	Горячие клавиши, нажатие которых вызывает данное действие. В Инспекторе объектов их можно выбрать из <u>выпадающего списка</u> , или набрать с клавиатуры. Возможны варианты из одной клавиши, или комбинации с клавишами Alt, Ctrl и Shift, записанные через '+', например: Ctrl+A, Alt+Ctrl+D, Alt+Ctrl+Shift+F, F6, Del и т.п.
ImageIndex: TImageIndex	Номер иконки в списке ImageList. Иконку можно выбрать из выпадающего списка, но вначале ее нужно добавить в ImageList. Если ImageIndex = -1, то иконки нет. <i>Для использования иконок, необходимо связать ActionList и ImageList</i>
Visible: Boolean;	Видимость связанных с Действием кнопок и пунктов меню
Enabled: Boolean;	Доступность связанных с Действием кнопок и пунктов меню. При Enabled := False, текст и изображение на компоненте становятся <u>серыми</u> . <i>Многие стандартные Action-ы сами управляют своей доступностью, например, если в буфере обмена нет текста, то кнопка «Вставить» («Paste») будет недоступна</i>
AutoCheck: Boolean;	Для пункта меню позволяет переключаться между выбранным («галочка» установлена) и невыбранным состоянием («галочка» снята). Это же свойство позволяет кнопке переключаться между «зажатым» и «отпущенным» положением
Checked: Boolean;	Дополняет предыдущее свойство, и позволяет установить «галочку» для пункта меню или зажать кнопку
GroupIndex: Integer;	Номер группы. Объединяет несколько действий (с одинаковым номером) в одну группу. В группе одновременно может быть нажата только одна кнопка (или выбран только один из пунктов меню). Если мы нажимаем другую кнопку, то предыдущая «отщелкивается». Когда GroupIndex = 0, то группировки нет

Свойства Action-ов со встроенными диалогами немного отличаются (рис. 55), для них нет свойств **AutoCheck**, **Checked** и **GroupIndex**, но появляется свойство **Dialog**, которое можно развернуть. Внутри будет один из тех самых диалогов, которые были рассмотрены ранее (в разделах 2.6 и 2.7).

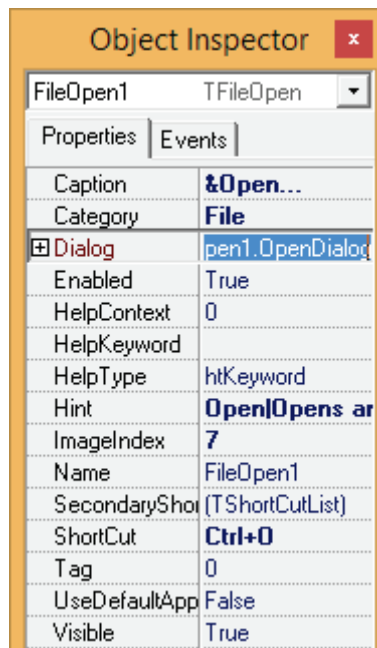


Рисунок 55 – Действие со встроенным диалогом

Кроме того, существуют **Action**-ы с привязками (для самостоятельного углубленного изучения подробнее см. раздел 2.10).

2.9. РАБОТА № 2 «Разработка текстового редактора»

Задание

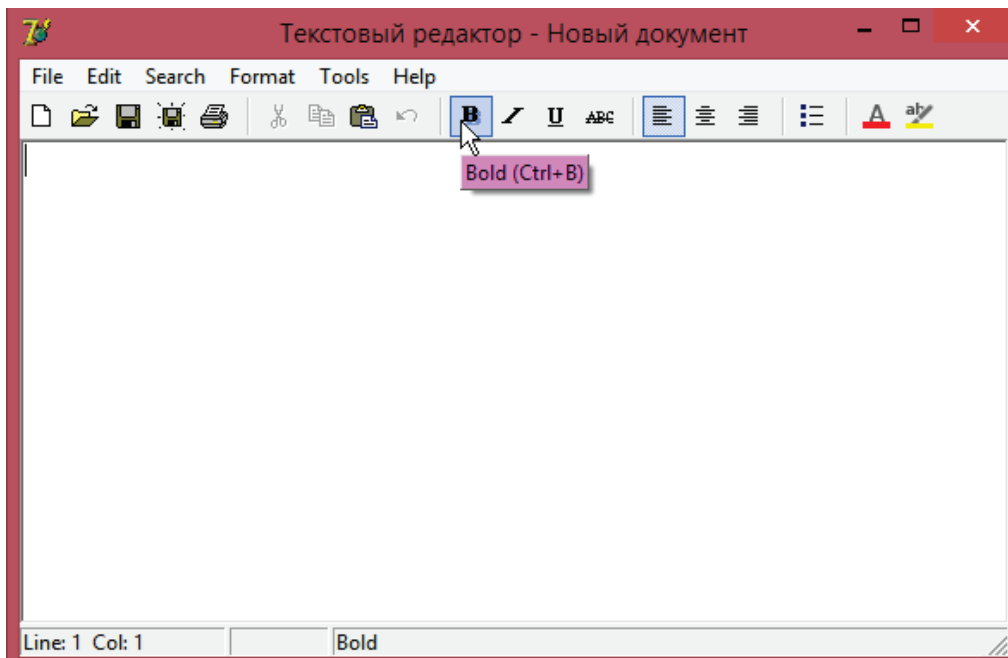


Рисунок 56 – Внешний вид текстового редактора

Разработать текстовый редактор (упрощенную версию Word), приблизительный внешний вид которого представлен на рисунке 56. Редактор должен содержать как минимум одну панель управления с кнопками, главное

меню (сверху) и всплывающее меню. Для кнопок должны отображаться всплывающие подсказки (см. «Bold (Ctrl+B)» на рисунке 56). Все названия (Caption) и всплывающие подсказки (Hint) должны быть переведены на русский язык. Для основных функций должны быть задействованы «горячие клавиши» (такие, например, как «Ctrl+C»).

Все действия для кнопок и пунктов меню добавляются исключительно через **ActionList** (либо через **ActionManager**, при желании выполнить работу сверх задания). Весь функционал, для которого имеются готовые стандартные Action-ы (действия), должен быть реализован через них. Для остальных действий необходимо самим создать новые **Action-ы**, а также подготовить соответствующие изображения (иконки) для кнопок (и пунктов меню).

Данная часть работы не подразумевает написания ни единой строки кода, вся работа производится в графической части среды разработки **Delphi**. При этом будут работать не все функции редактора, например, при нажатии кнопок «Открыть» или «Сохранить как» будут отображаться диалоговые окна выбора файлов, а фактического сохранения производиться не будет. *Но при желании созданный в редакторе текст всегда можно скопировать в буфер обмена и вставить в Word.*

Функции программы

Разрабатываемая программа должна выполнять как минимум следующие функции:

- открытие и сохранение документа, создание нового документа, закрытие программы;
- печать документа, диалоговые окна настройки принтера и страницы;
- работа с буфером обмена (вырезать, копировать, вставить, выделить все, удалить);
- отмена последнего изменения;
- поиск и замена текста;
- выделение текста **жирным** и *курсивом*, подчеркивание и зачеркивание текста;
- выравнивание текста по левому и правому краям, а также по центру;
- создание списков;
- диалоговые окна настройки шрифта и цвета текста;
- кнопка изменения цвета текста на красный;
- кнопка выделения текста желтым маркером, т.е. желтым фоном (не обязательная функция, выполняется по желанию);
- настройка панелей управления во время работы программы (не обязательно, только для случая с **ActionManager**);
- пункт меню «О программе» (About), который в дальнейшем будет отображать окно с именем автора, годом разработки и прочей информацией о программе.

Пример тестирования работы с форматом текста представлен на рисунке 57.

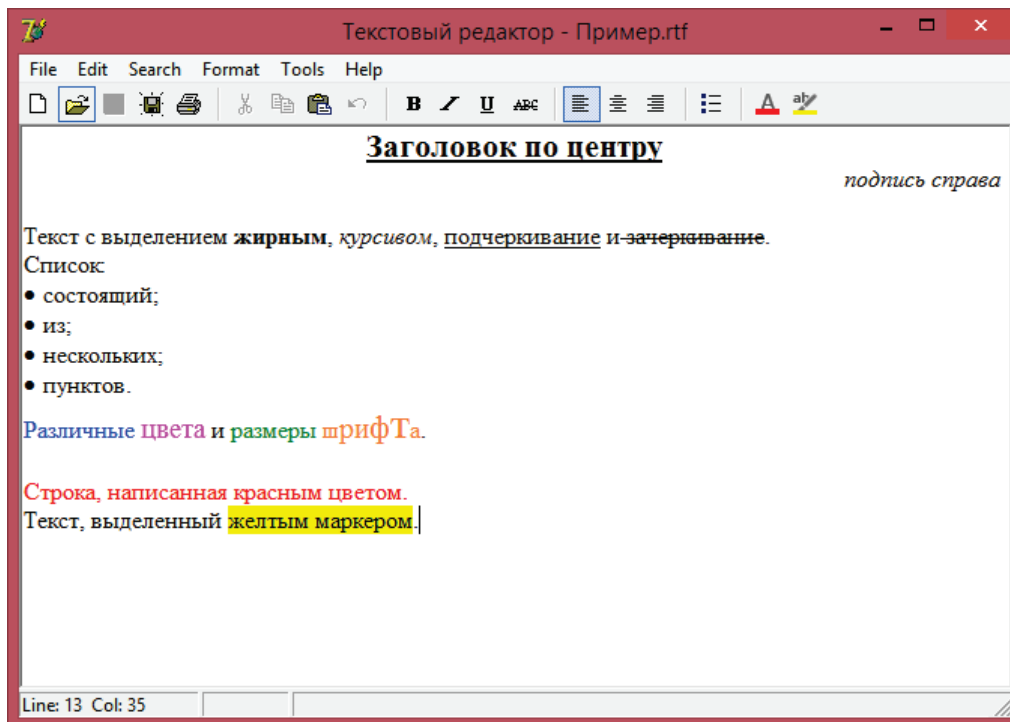


Рисунок 57 – Форматирование текста

Меню программы

Пример главного меню программы представлен на рисунке 58.

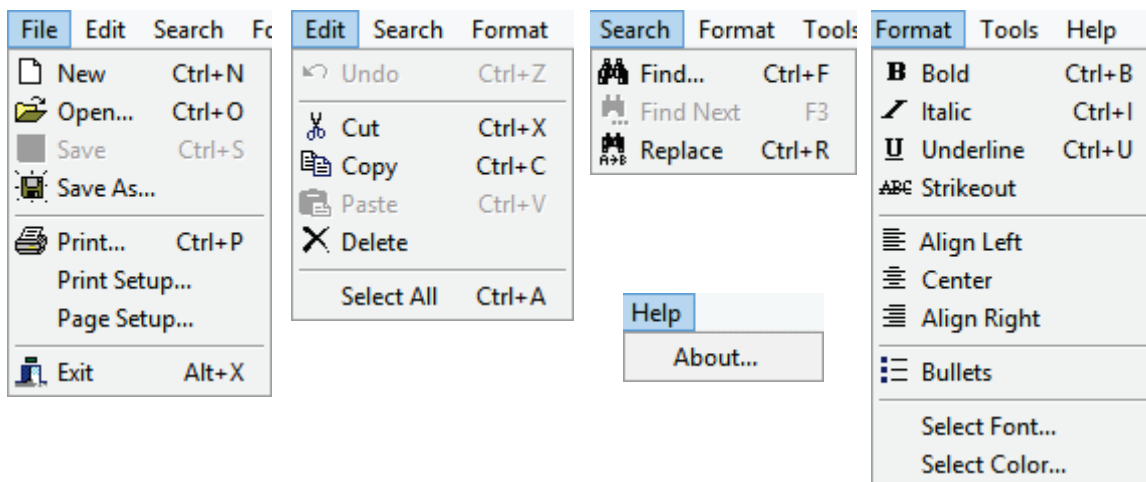


Рисунок 58 – Главное меню программы

В случае если вместо **ActionList** применяется **ActionManager**, то дополнительно используется пункт меню для настройки панели (рис. 59).

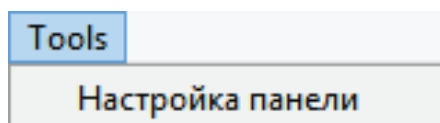


Рисунок 59 – Настройка панелей ActionManager

Пример всплывающего меню представлен на рисунке 60.

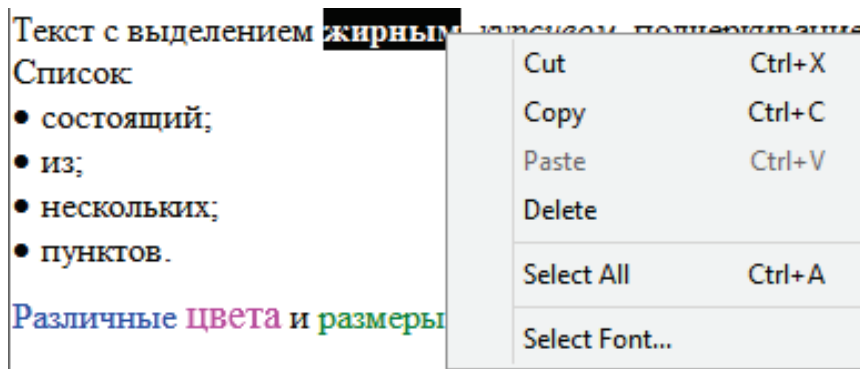


Рисунок 60 – Всплывающее меню

Начало работы

Добавить на форму следующие компоненты (со вкладок «**Standard**» и «**Win32**»):

- **RichEdit** (задать ему свойство `Align = alClient`);
- **StatusBar**;
- **ImageList**;
- **ActionList** (связать его с ранее добавленным **ImageList**, задав свойство `Images = ImageList1`);
- **ToolBar** (связать его с **ImageList**).

*Если вместо **ActionList** используется **ActionManager**, то вместо **ToolBar** необходимо использовать **ActionToolBar**.*

- **MainMenu**;
- **PopupMenu** (связать данное меню с **RichEdit**, задав свойство `RichEdit.PopupMenu = PopupMenu1`).

Далее, зайдя в компоненте **ActionList**, добавить стандартные Action-ы:

- **TFileOpen**, **TFileSaveAs**, **TFilePrintSetup**, **TFilePageSetup**, **TFileExit** из категории «File»;
- **TEditCut**, **TEditCopy**, **TEditPaste**, **TEditSelectAll**, **TEditUndo**, **TEditDelete** из категории «Edit»;
- **TSearchFind**, **TSearchFindNext**, **TSearchReplace** из категории «Search»;
- **TRichEditBold**, **TRichEditItalic**, **TRichEditUnderline**, **TRichEditStrikeOut**, **TRichEditBullets**, **TRichEditAlignLeft**, **TRichEditAlignRight**, **TRichEditAlignCenter** из категории «Format»;
- **TColorSelect**, **TFontEdit**, **TPrintDlg** из категории «Dialog»;
- **TCustomizeActionBars** из категории «Tools» (только для случая с **ActionManager**, не применяется при использовании **ActionList**).

На панели **ToolBar** нажать правую клавишу мышки и добавить необходимое количество кнопок через пункт меню «New Button» (рис. 61). Для добавления «разделителей» между кнопками, используется пункт меню «New Separator».

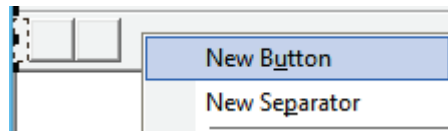


Рисунок 61 – Добавление кнопок на панель ToolBar

Далее выделяем первую из кнопок и переходим в «Инспекторе объектов» (рис. 62). Для свойства **Action** из выпадающего списка выбираем «FileOpen1».

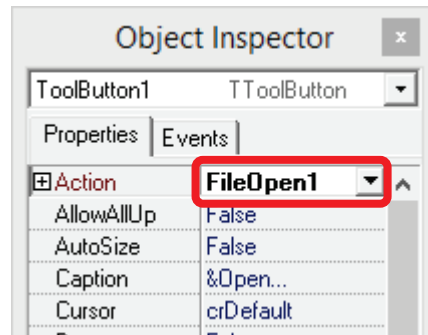


Рисунок 62 – Привязка действия к кнопке

После привязки, кнопка на панели примет следующий вид (рис. 63).



Рисунок 63 – Кнопка после привязки

Если сейчас запустить программу (F9) и нажать на данную кнопку, то отобразится диалоговое окно «Открыть» (рис. 64). Хотя к настоящему моменту мы не написали ни единой строчки кода.

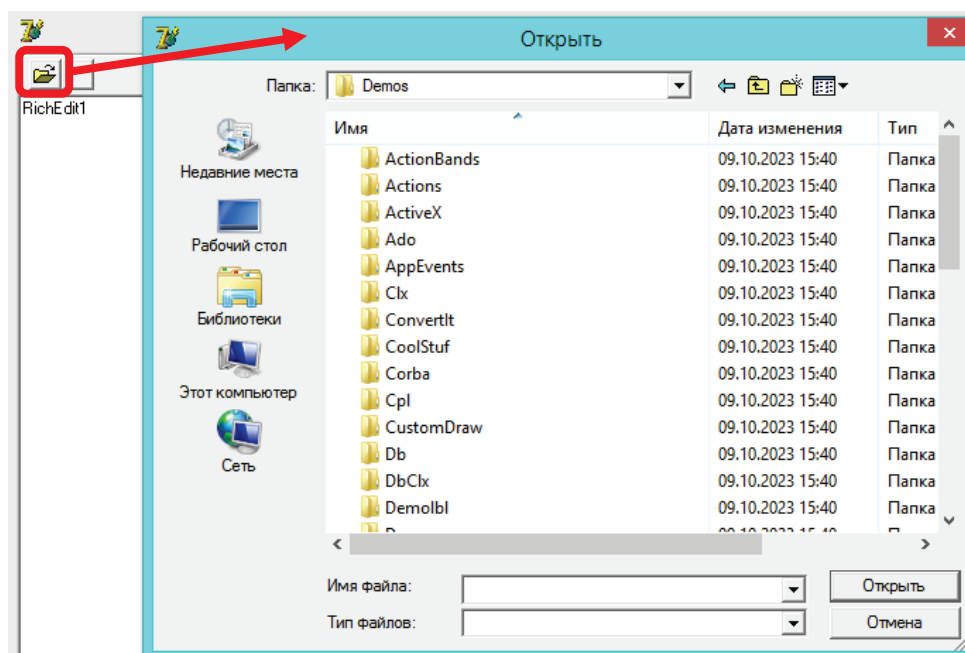


Рисунок 64 – Диалоговое окно «Открыть»

Продолжение работы

Аналогичным образом настраиваем **Action**-ы для всех кнопок. На данном этапе панель управления должна выглядеть как на рисунке 65. Конечно, пока у нас имеются не все требуемые кнопки, например, нет кнопок «New» и «Save», т.к. для них не существует стандартных **Action**-ов. Но уже сейчас можно протестировать, как наше приложение умеет форматировать текст.



Рисунок 65 – Стандартные Action-ы на панели управления

Далее аналогично привязываем все **Action**-ы к пунктам Главного меню, а также заполняем всплывающее меню. Для добавления «разделителей» в меню, нужно задать им свойство `Caption := '-'` (минус).

После этого открываем **ActionList** и переводим все надписи и всплывающие подсказки на русский язык.

Что дальше?

Для тех кнопок (и пунктов меню), для которых не нашлось стандартных **Action**-ов, необходимо создать свои **Action**-ы. Кроме того, для них нужно самим подготовить иконки (изображения в формате *.bmp).

Кнопки с собственными Действиями (такие как «New» и «Save») будут недоступны, т.е. иметь серый цвет (рис. 66) – на данном этапе это нормально, т.к. мы еще не написали код для этих действий. Некоторые отрывки кода, которые необходимо использовать в данной работе, будут приведены далее.



Рисунок 66 – Недоступные кнопки

2.10. *Action-ы с привязками

Action-ы, с которыми мы работали ранее (такие, например, как **TEditCopy**, или **TRichEditBold**), сами определяли с каким компонентом им работать. Но для некоторых стандартных Действий, требуется явно указать компонент, с которым они должны взаимодействовать.

Рассмотрим стандартные Действия **TPreviousTab** и **TNextTab** (рис. 67). Они позволяют переключать («Next» – следующая, «Previous» – предыдущая) вкладки для компонентов **PageControl** и **TabControl**.

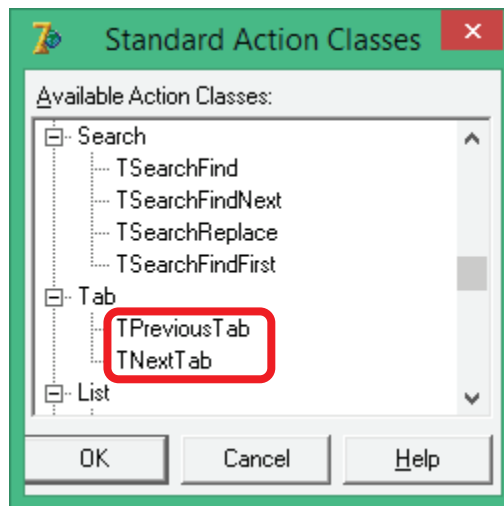


Рисунок 67 – Действия, переключающие вкладки

В Инспекторе объектов для этих **Action**-ов имеется свойство **TabControl** (рис. 68), в котором необходимо выбрать компонент **PageControl** или **TabControl**.

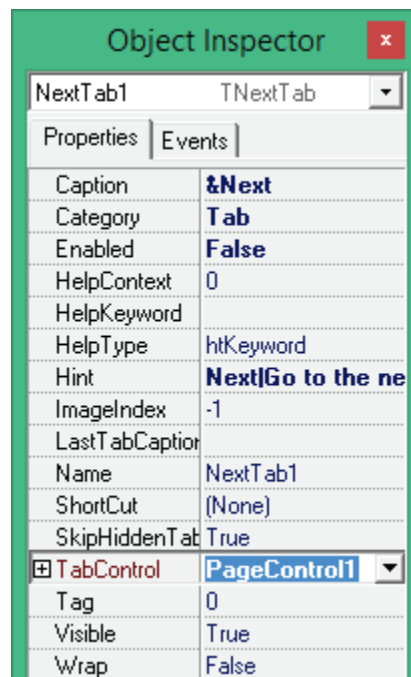


Рисунок 68 – Привязка действия

Для этого добавим на форму компонент **PageControl** (со вкладки «Win32») и, нажав на него правой клавишей мышки, добавим несколько вкладок «**New Page**» (рис. 69).

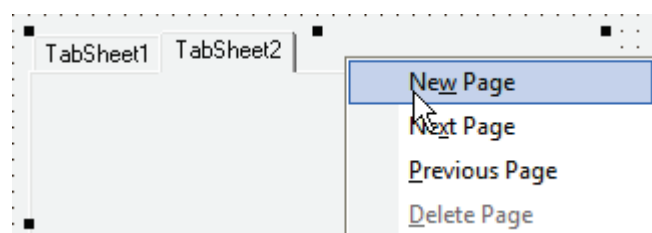


Рисунок 69 – Добавление вкладок

Также добавим две кнопки, которые свяжем с нашими новыми Action-ами. Запустим и протестируем работу кнопок (рис. 70).

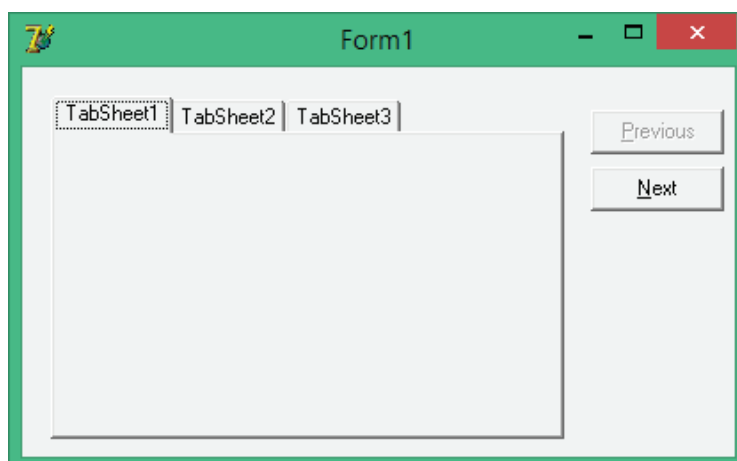


Рисунок 70 – Тестирование действий

Кроме привязки, рассмотренные Действия содержат Свойство «**Wrap**» (рис. 68), включение которого позволяет осуществить переход к следующей вкладке «с прокруткой» (т.е. после достижения последней вкладки, мы снова попадаем на первую).

Существуют и другие стандартные Действия с привязкой к Компонентам.

2.11. *ActionManager

Вместо **ActionList** можно использовать его продвинутый вариант **ActionManager**. Обычно он применяется совместно с **ActionMainMenuBar** и **ActionToolBar**. Все эти компоненты расположены на вкладке «**Additional**» (рис. 71).

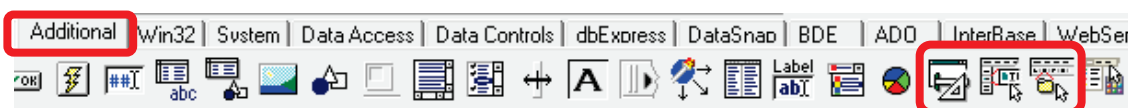


Рисунок 71 – Вкладка Additional

Так же, как это было для **ActionList**, **ActionManager** необходимо связать с **ImageList**.

Можно выделить 3 уровня использования **ActionManager**, начиная с нулевого уровня.

Уровень 0

На этом уровне **ActionManager** используется совершенно точно также, как и **ActionList**. Все компоненты, для которых задаются действия, как и раньше создаются вручную и по одному связываются со своими Action-ами.

От **ActionManager** используется только вкладка «**Actions**» (рис. 72). Остальные вкладки на Уровне 0 не применяются. Такой вариант не дает никаких преимуществ по сравнению с использованием **ActionList**, но закладывает потенциал для будущего перехода на Уровни 1 и 2.

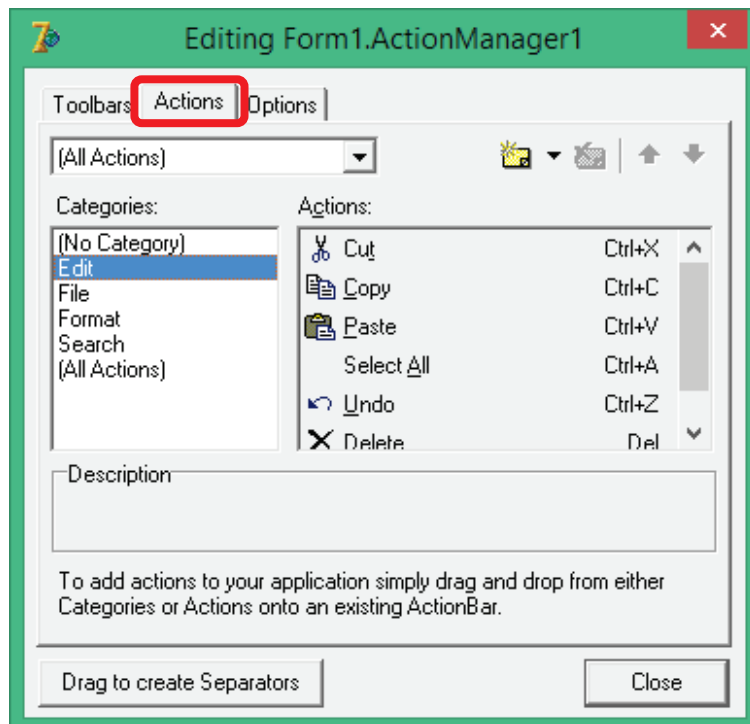


Рисунок 72 – Раздел Actions

Уровень 1

В этом случае необходимо добавить на Форму компонент главного меню **ActionMainMenuBar** и одну или несколько панелей **ActionToolBar** со вкладки «Additional» (рис. 71). Данные компоненты добавляются вместо компонентов **MainMenu** и **ToolBar**.

После этого, мы получаем возможность перетаскивать Действия мышкой из окна **ActionManager** прямо на соответствующие панели (рис. 73). При этом мы можем перетаскивать как отдельные действия, так и целые Категории действий.

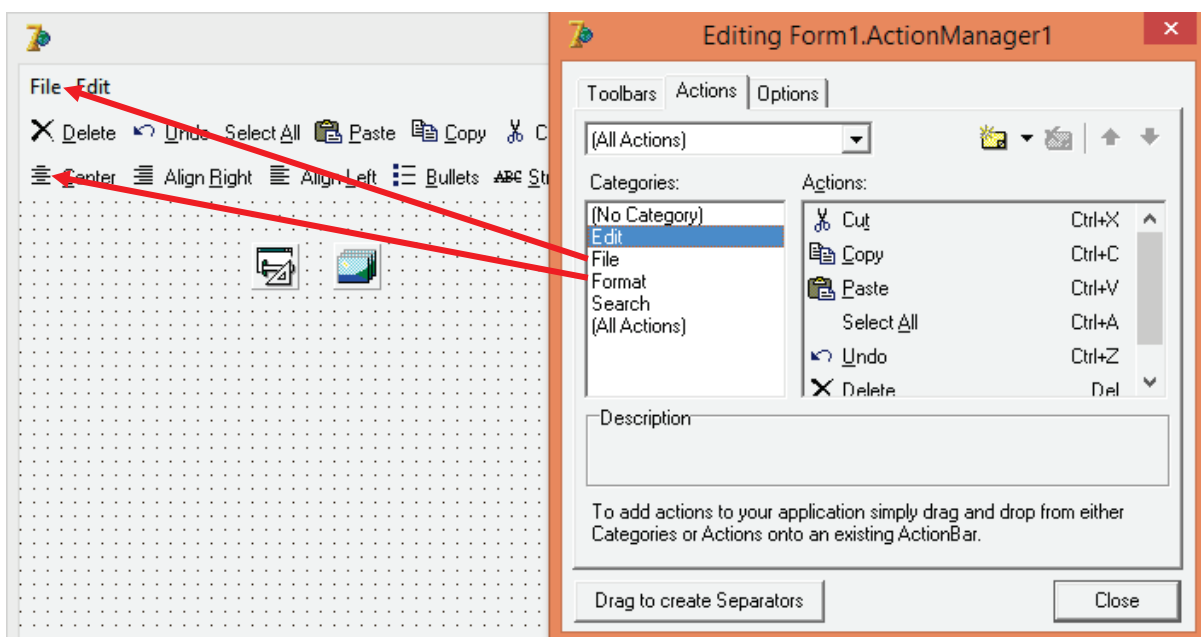


Рисунок 73 – Добавление кнопок и пунктов меню

Можно перемещать кнопки (и пункты меню) внутри панели или между панелями при помощи мышки. Для удаления кнопки достаточно перетащить ее за границы панели.

Также можно перетаскивать мышкой Разделители «**Separators**» (рис. 74) для пунктов меню и панелей с кнопками.

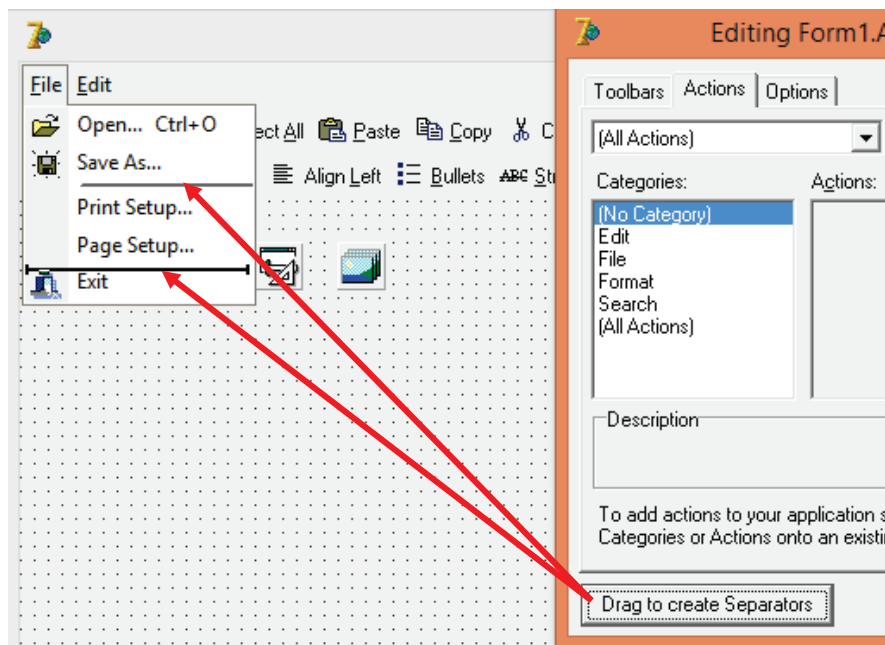


Рисунок 74 – Добавление разделителей

Теперь нам доступны и другие вкладки в окне **ActionManager**. На вкладке «**Toolbars**» (рис. 75) будут отображены все добавленные нами панели. Панели можно отключать (скрывать), сняв соответствующие галочки. Также здесь можно настроить, будет ли для кнопок отображаться текстовая подпись, или только соответствующая картинка. *Стоит обратить внимание, что если для кнопки не задана картинка, то скрыть текстовую подпись не получится.*

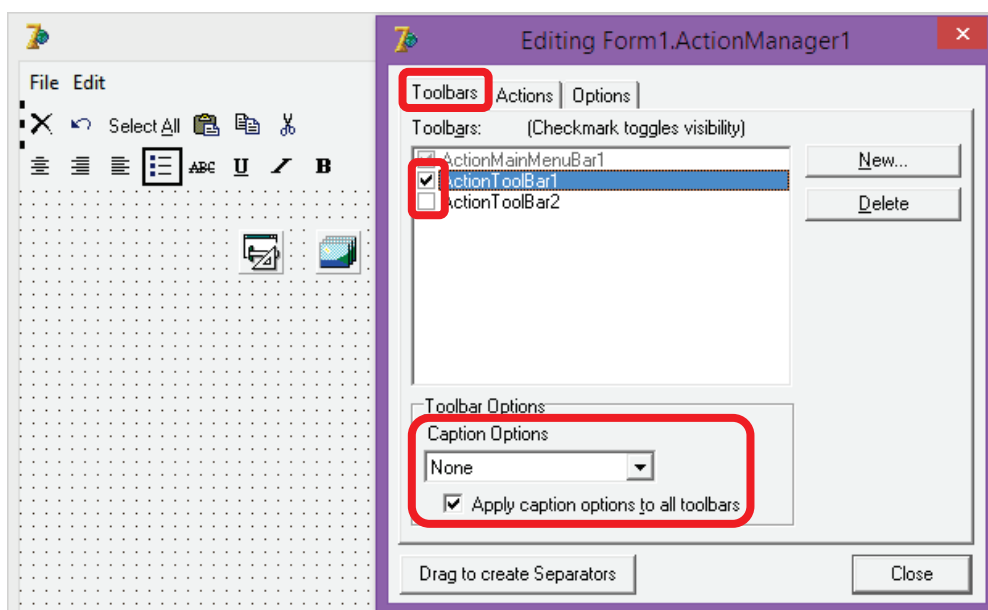


Рисунок 75 – Настройка панелей

Такой вариант позволяет значительно быстрее разрабатывать панели и меню на основе Действий. Но у него есть и недостаток: в такую панель невозможно добавить ничего кроме **Action**-ов, в то время как на обычную панель **ToolBar** мы можем добавить не только кнопки, но и поля ввода **Edit** и **SpinEdit**, выпадающие списки **ComboBox** и многое другое. В этом случае придется использовать обычный **ToolBar**.

Дизайн и стили

Панель **ActionToolBar** позволяет задать разные варианты отображения (рис. 76).

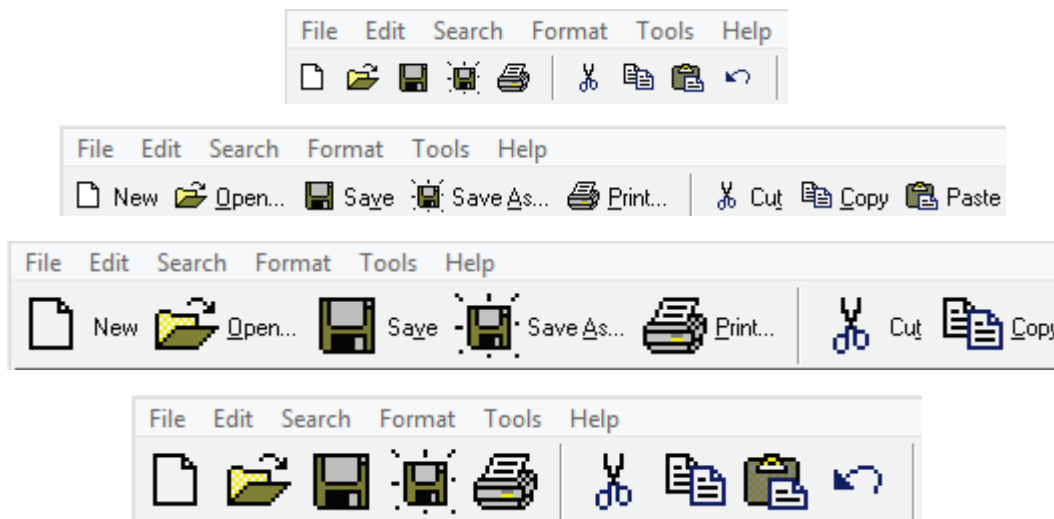


Рисунок 76 – Варианты отображения панели ActionToolBar

Эти настройки можно произвести на вкладке «**Toolbars**» (рис. 75) и на вкладке «**Options**» (рис. 77).

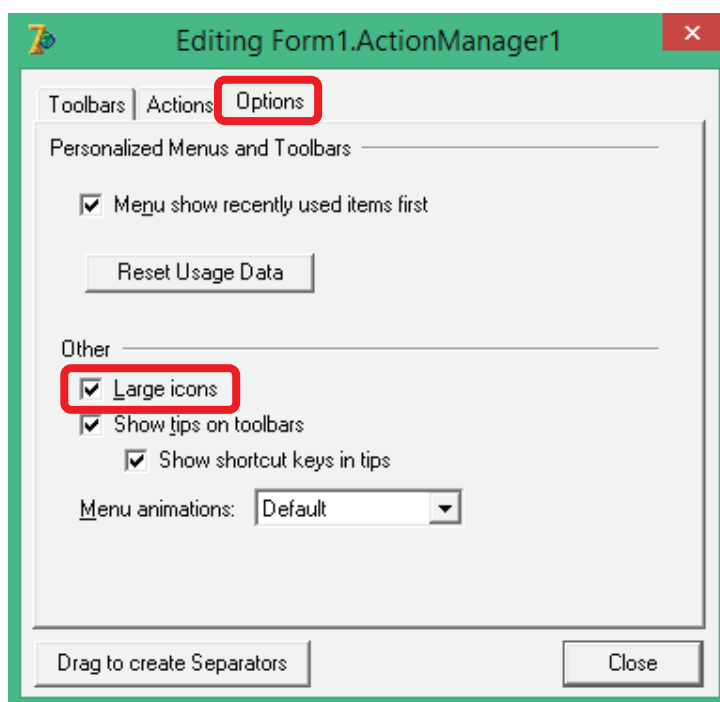


Рисунок 77 – Отображение больших иконок

Кроме того, для **ActionToolBar** и **ActionMainMenuBar** можно задать стили отображения **ColorMap** (рис. 78).

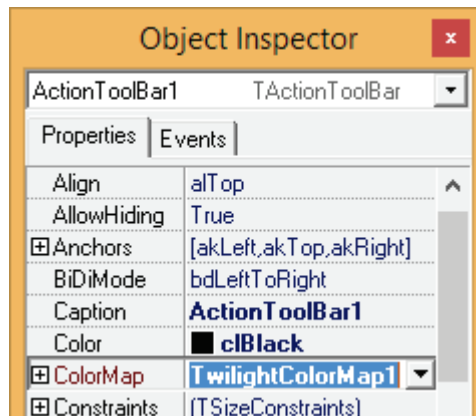


Рисунок 78 – Выбор стиля отображения

Невизуальные компоненты, отвечающие за стили, расположены на вкладке «**Additional**» (рис. 79). Имеются стили **TStandardColorMap**, **TXPColorMap**, а также черно-белая тема **TTwilightColorMap**. Каждый стиль содержит настройки для множества различных цветов (рис. 80).



Рисунок 79 – Добавление стилей

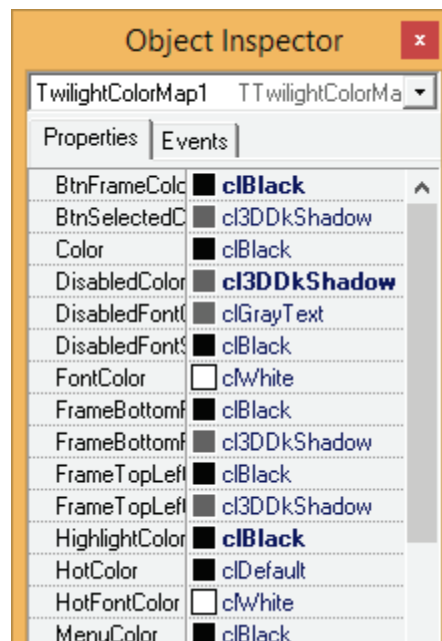


Рисунок 80 – Настройка цветов для стиля

Стоит обратить внимание, что стандартные иконки не предназначены для работы на черном фоне (рис. 81), и их потребуется перерисовать вручную.



Рисунок 81 – Неверное отображение на черном фоне

Либо можно указать не чисто черный цвет, а заменить его на какой-то из темных (рис. 82). Для этого в свойстве **ColorMap** необходимо изменить цвета, например:

```
Color := clMaroon;  
MenuColor := clMaroon;
```



Рисунок 82 – Темный фон

Уровень 2

ActionManager не только позволяет менять кнопки на панелях при проектировании приложения разработчиком, но и предоставляет возможность перемещать, удалять и добавлять кнопки пользователю итоговой программы.

Для этого добавим действие **TCustomizeActionBars** (рис. 83). После чего перетащим его в главное меню.

!!! Для того, чтобы диалоговое окно редактирования панелей всегда оставалось поверх главного окна приложения, рекомендуется также включить:

```
CustomizeActionBars1.CustomizeDlg.StayOnTop := True;
```

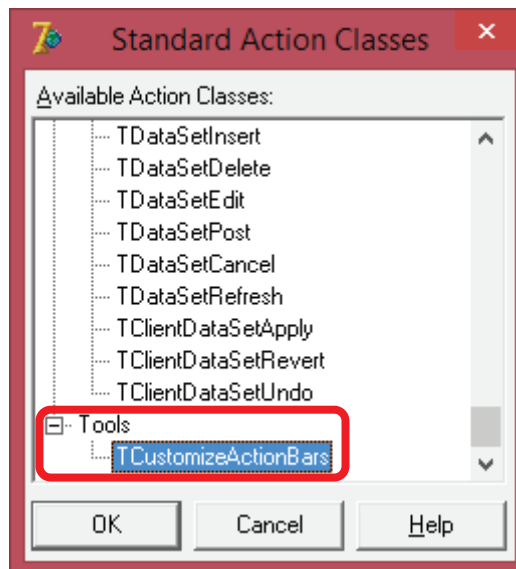


Рисунок 83 – Добавление действия Customize

На этом уровне необходимо также задать имя файла **FileName** для сохранения настроек, например, «ActionManager.settings» (рис. 84).

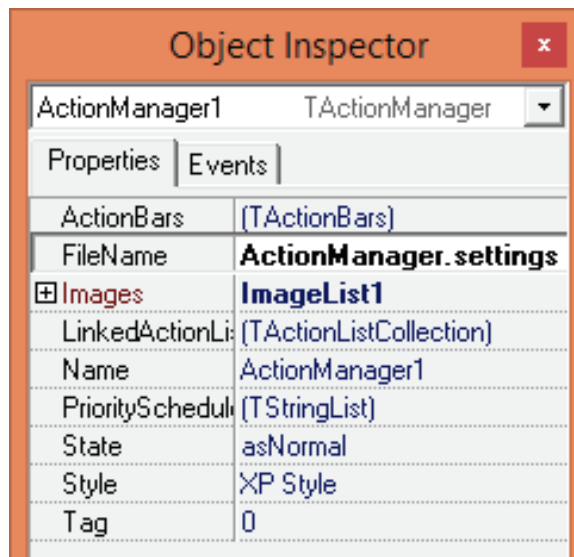


Рисунок 84 – Настройка имени файла

!!! Для того чтобы сбросить настройки пользователя, необходимо удалить данный файл «ActionManager.settings» из папки с программой. *Лучше задавать имя файла после окончания разработки проекта, а на время проектирования очищать этот параметр. В противном случае после компиляции (F9) очередной версии проекта, мы будем видеть старую версию панелей и меню.*

Недостатком данного случая является то, что окно **ActionManager** существует только на английском языке.

Смешанный вариант

Также можно применять смешанный вариант, например, использовать панель **ActionToolBar** совместно с обычным главным меню **MainMenu**. В этом случае каждый пункт меню придется привязывать к **Action**-ам вручную.

2.12. Событийно-ориентированное программирование

Событийно-ориентированное программирование (event-driven programming), или иногда кратко, «**Событийное программирование**» – парадигма программирования, в которой выполнение программы определяется событиями (**Events**) – т.е. действиями пользователя (клавиатура, мышь, сенсорный экран), а также сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета). Событийно-ориентированное программирование применяется при построении пользовательских интерфейсов, или, например, при создании игр, в которых осуществляется управление множеством объектов.

Несмотря на то, что **Delphi** поддерживает различные парадигмы, включая Модульное программирование (*с которым мы уже сталкивались*) или Объектно-ориентированное программирование (*с которым мы еще встретимся дальше*), основой **Delphi**, значительно упрощающей жизнь программисту, является именно Событийно-ориентированное

программирование, которое позволяет реализовать концепцию «быстрой разработки приложений» (**RAD**).

На самом деле, мы уже сталкивались с событиями (*хотя сами этого и не замечали*), это было событие **Click** («Щелчок») для кнопки. Когда в графическом редакторе Delphi мы кликали мышкой дважды по кнопке, то у нас автоматически создавалась новая процедура, имеющая вид:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    //Здесь писался код для кнопки
end;
```

После запуска программы (**F9**), если мы осуществляли клик мышкой по этой кнопке, то выполнялся соответствующий код.

Внутри подобных процедур пишется код, который выполнялся «если» была нажата кнопка, но есть одно «но», у нас в коде нигде не написано команды «**if**», которая переводится как «**если**»... Поэтому мы не должны говорить, «Если нажата кнопка».

Представим на секунду, что у нас не существует никаких событий, есть только кнопка Button1, программа, повторяемая в бесконечном цикле, и три функции:

```
X: Integer;
Y: Integer;
LeftBth: Boolean;
```

Первые две позволяют получить текущие координаты мышки, а LeftBth сообщает нам, нажата ли кнопка мышки в настоящий момент, или нет.

Тогда прежде чем написать код для кнопки, нам вначале самостоятельно нужно проверить, нажата ли кнопка мышки, а если нажата, то нужно определить еще, что она нажата именно внутри этой кнопки Button1. Такой код будет иметь следующий вид:

```
if LeftBth then
begin
    if (X >= Button1.Left) and (X < Button1.Left + Button1.Width) and
        (Y >= Button1.Top) and (Y < Button1.Top + Button1.Height) then
        begin
            //Код для кнопки
        end;

    //...
end;
```

Так много лишнего нам пришлось написать всего для одной кнопки. *Причем в конце мы не зря оставили многоточие, туда нужно будет писать код для других кнопок... Сколько их было для «Калькулятора»?*

Кроме того, на самом деле это даже неполный код. В приведенном коде при удерживании кнопки мышки команда будет выполняться многократно в цикле, но нам нужно чтобы на каждое нажатие мышки, команда выполнялась только один раз (причем неважно сколько мы удерживаем кнопку). *Мы не будем даже пытаться доработать код до этого усложненного варианта.*

К счастью в **Delphi** все это писать и не нужно, а создаваемые средой процедуры правильно называть не «Если нажата кнопка», а «Когда нажата

кнопка». А еще точнее «по» или «после», например, «По щелчку мышкой» или «После движения мышки».

В действительности в **Delphi** осуществляются все те проверки координат, которые мы рассмотрели, но нам их писать не нужно, т.к. они уже реализованы разработчиками **Delphi**, и названы событием **OnClick**, которым мы и пользовались ранее. В этом и заключается смысл Событийно-ориентированного программирования.

События для компонентов можно найти в Инспекторе объектов, но нужно перейти на вкладку «Events» (рис. 85). Имена событий в **Delphi** принято начинать с приставки «On», которая и переводится как «по» или «после». Например, **OnClick** – «По щелчку», или **OnMouseMove** – «После движения мышки».

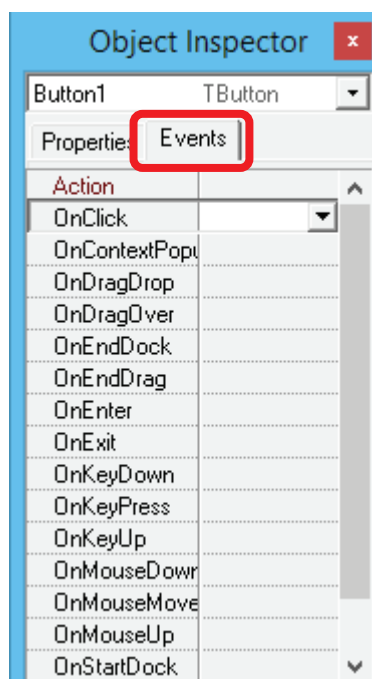


Рисунок 85 – Пример событий для кнопки

Для использования требуемого события необходимо дважды кликнуть по нему мышкой в Инспекторе объектов. При этом для данного события будет создан требуемый код (называемый «**Обработчик события**»), а курсор ввода будет помещен между соответствующими **begin** и **end**, что позволяет сразу начать вводить команды, выполняемые при данном событии. *То же самое мы уже видели, когда дважды кликали по **Button**-ам при создании «Калькулятора».* После этого в Инспекторе объектов для данного события отобразится имя его обработчика. При желании можно его переименовать (*но обычно этого делать не рекомендуется*). Кроме того, в Инспекторе объектов можно выбирать Обработчики событий из выпадающего списка (если они ранее уже были созданы), это позволяет привязать один обработчик событий сразу к нескольким компонентам.

2.13. События визуальных компонентов

Основные события визуальных компонентов представлены в таблице

25. Таблица 25 – Основные события визуальных компонентов

Событие	Описание
OnClick: TNotifyEvent;	Клик («щелчок») мышкой по компоненту. У кнопки Button это событие также срабатывает, если нажать «Enter» или «Пробел» в тот момент, когда фокус ввода находится на кнопке (<i>обычно при этом кнопка выделена пунктирной рамкой</i>)
OnDblClick: TNotifyEvent;	Двойной клик мышкой. <i>Имеется не у всех компонентов</i>
OnContextPopup: TContextPopupEvent;	Всплывающее меню. Фактически это нажатие правой клавиши мышки. В отличие от OnClick и OnDblClick, возвращает еще и координаты нажатия мышки. <i>Координаты для данного события доступны по именам MousePos.X и MousePos.Y</i>
OnMouseMove: TMouseMoveEvent;	Движение курсора мышки над компонентом (при этом нажатие клавиш мышки не обязательно, хотя и возможно). Возвращает координаты курсора мышки. Также возвращает информацию о том, нажата ли какая-то клавиша мышки, и удерживаются ли при этом клавиши Alt, Ctrl или Shift
OnMouseDown: TMouseEvent; OnMouseUp: TMouseEvent;	Нажатие или отпускание клавиши мышки (не обязательно левой). Это более подробный случай для OnClick. Позволяет разделить клик мышкой на нажатие и отпускание клавиши. Возвращает координаты курсора, а также информацию о том, какая клавиша мышки нажата и удерживаются ли при этом клавиши Alt, Ctrl или Shift
OnKeyPress: TKeyPressEvent;	Нажатие клавиши на клавиатуре. Возвращает символ нажатой клавиши. <i>Фокус ввода должен находиться на компоненте (обычно это отображается как мигающий курсор ввода текста или как пунктирная рамка вокруг кнопки)</i>

Событие	Описание
OnKeyDown: TKeyEvent; OnKeyUp: TKeyEvent;	Нажатие или отпускание клавиши на клавиатуре. Это более подробный случай для OnKeyPress. Позволяет разделить нажатие клавиши клавиатуры на нажатие и отпускание. Возвращает код нажатой клавиши, а также информацию о том, удерживаются ли при этом клавиши Alt, Ctrl или Shift
OnEnter: TNotifyEvent; OnExit: TNotifyEvent;	Срабатывает, когда данный компонент получает фокус или теряет его. Фокус ввода отображается как мигающий курсор ввода текста или как пунктирная рамка вокруг выбранной кнопки. Для переключения фокуса компонентов помимо мышки также можно использовать кнопку «Tab» (но кнопка «Tab» не будет работать для компонента, для которого установлено значение TabStop := False). Только один компонент на форме может иметь фокус
OnChange: TNotifyEvent;	«Изменение». Срабатывает при изменении содержимого компонента. Например, при изменении текста в Edit, Memo или RichEdit или изменении численного значения в SpinEdit
OnSelectionChange: TNotifyEvent;	Срабатывает при изменении положения каретки (курсора для ввода текста). <i>Есть только у RichEdit</i>
OnMouseWheel: TMouseWheelEvent; OnMouseWheelDown: TMouseWheelUpDownEvent; OnMouseWheelUp: TMouseWheelUpDownEvent;	Вращение колесика мышки. Есть далеко не у всех компонентов. Имеется, например, у RichEdit. <i>При этом компонент должен быть в фокусе, т.е. курсор ввода текста должен быть на нем</i>
OnMouseEnter: TNotifyEvent; OnMouseLeave: TNotifyEvent;	Срабатывают, когда курсор мышки заходит в область над компонентом или выходит из нее. <i>В старых версиях (таких как Delphi 7) они скрыты почти для всех компонентов, но в новых версиях (таких как Delphi XE8) доступны для большинства компонентов</i>

Уведомления

Самым простым типом для событий являются «Уведомления» (**Notify** – «Уведомить», «Поставить в известность»):

TNotifyEvent = procedure(Sender: TObject) of object;

Такое событие не передает никаких дополнительных данных, кроме адреса отправителя (**Sender**). Для него Delphi генерирует следующий обработчик:

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

end;
```

!!! Стоит обратить внимание, что в создаваемом автоматически имени такой процедуры всегда будет указано имя события (в данном случае «**Click**»), а перед ним – имя компонента, для которого наступает это событие (в данном случае это «**Button1**»).

Движение мышки

```
TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState;
                             X, Y: Integer) of object;
```

Данный вид события не только «уведомляет» о том, что что-то произошло, но и сообщает, где это произошло, передавая координаты курсора.

Например, отобразим в заголовке окна, координаты курсора мышки, который двигается над формой TForm1. Для этого в обработчике события **OnMouseMove** этой формы напишем:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
                               X, Y: Integer);
begin
  Caption := IntToStr(X) + ':' + IntToStr(Y);
end;
```

Координаты отсчитываются от верхнего левого угла окна (рис. 86).

!!! Этот пример с **OnMouseMove** стоит запомнить, т.к. он (как и следующие примеры с мышкой) уже скоро понадобится нам при разработке «Графического редактора».



Рисунок 86 – Координаты курсора в заголовке

Также данное событие сообщает нам параметр **Shift**, типа:

TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

где ssShift, ssAlt, ssCtrl – сообщают, удерживаются ли в данный момент клавиши Shift, Alt или Ctrl;

ssLeft, ssRight, ssMiddle – сообщают, удерживаются ли в данный момент какие-то из клавиш мышки (левая, правая или средняя);

ssDouble – сообщает, что был двойной клик мышкой.

Стоит обратить внимание, что **TShiftState** объявлен как **SET** («Набор»). К нему мы обращаемся не через «равенство» или «точку», здесь необходимо использовать оператор **in** («в»), т.е. мы проверяем «входит ли значение в набор?».

Например, опять будем отображать координаты курсора при движении мышки, но теперь будем обновлять текст в заголовке только когда удерживается клавиша **Alt**:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
Y: Integer);
begin
  if ssAlt in Shift then
  begin
    Caption := IntToStr(X) + ':' + IntToStr(Y);
  end;
end;
```

Нажатие кнопок мышки

TMouseEvent = procedure(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer) of object;

Например, отобразим координаты нажатия кнопки мышки на форме:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
  Caption := IntToStr(X) + ':' + IntToStr(Y);
end;
```

Аналогично отобразим координаты отпускания мышки, но будем добавлять их к уже имеющимся в заголовке начальным координатам (рис. 87):

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
  Caption := Caption + ' -> ' + IntToStr(X) + ':' + IntToStr(Y);
end;
```

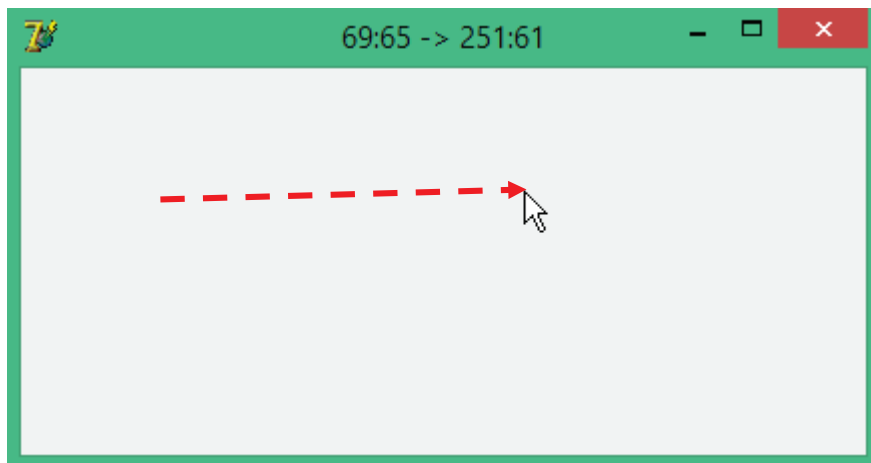


Рисунок 87 – Координаты отпускания курсора

Хорошая новость: на самом деле, нет необходимости выписывать все типы событий или заучивать их. Т.к. после двойного клика по требуемому событию, **Delphi** автоматически создаст для нас обработчик для этого события – процедуру, в заголовке которой будут видны все передаваемые параметры. Нам нужно лишь помнить, какие события являются только уведомлениями («**Notify**»), а какие сообщают нам дополнительную информацию (не «**Notify**»).

Ограничение ввода символов с клавиатуры и замена символов

Обработчик событий **OnKeyPress** возвращает символ нажатой клавиши **Key**. Но при желании его можно изменить (до того, как он будет отображен). Например, можно заменять точки на запятые при вводе в **Edit**. Кроме того, можно вообще запретить некоторые символы, выводя вместо них:

Key := #0;

Рассмотрим следующий пример, позволяющий вводить в поле **Edit** только числа:

```
{----- Разрешить вводить только числа -----}
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    //Замена точки на запятую
    if Key = '.' then Key := ',';
    //Разрешить ввод только указанных символов
    if not (Key in ['0'..'9', ',', '-', '+', 'e', 'E', #8]) then Key := #0;
end;
```

Здесь **Key** сравнивается со списком значений. Запись вида:

Key in ['a', 'b', 'c']

равносильна следующей записи:

(Key = 'a') or (Key = 'b') or (Key = 'c')

!!! Кроме того, можно задавать диапазоны значений через «многоточие» (две точки!).

Стоит обратить внимание на то, что для ввода чисел, недостаточно только цифр и запятой, нам необходим, как минимум, еще «минус». Кроме того, числа можно вводить через 'E' (или 'e').

Символ с номером **#8** означает «**Backspace**». Если не включить его в список разрешенных, то мы потеряем возможность стирать последний символ.

Пример для OnChange

Событие **OnChange** наступает при изменении текста в поле **Edit**. Это может быть использовано для расчета значений, зависящих от этого поля, «на лету» (т.е. без необходимости нажимать кнопку расчета). Например, в Edit1 и Edit2 вводятся два числа, а в Edit3 выводится их сумма (рис. 88):

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
    Edit3.Text := IntToStr(StrToInt(Edit1.Text) + StrToInt(Edit2.Text));
end;
```

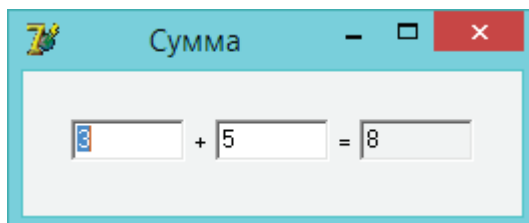


Рисунок 88 – Вывод результата «на лету»

Для Edit2 необходимо выполнять тот же код. Но его не нужно писать еще раз! Это тот самый случай, когда можно выбрать уже существующий обработчик из выпадающего списка (в Инспекторе объектов).

!!! Стоит обратить внимание, что в поля *Edit* можно ввести текст, который не может быть числом, что приведет к ошибке! Например, если ввести букву, то сразу выскочит сообщение об ошибке. Аналогично, ошибка произойдет и если полностью стереть текст ($Edit1.Text := ''$). Т.е. для правильной работы данной программы, ее необходимо дополнить условиями с проверками. Также можно ограничить вводимые в *Edit* символы, при помощи *OnKeyPress*.

Кроме того, можно использовать событие **OnChange** для изменения цвета текста, в зависимости от введенного в него значения, например:

```
procedure TForm1.Edit1Change(Sender: TObject);
var f: Double;
begin
    //Выделяем красным, если не является числом
    if TryStrToFloat(Edit1.Text, f) then Edit1.Font.Color := clBlack
    else Edit1.Font.Color := clRed;
end;
```

Пример для OnSelectionChange

Будем выводить координаты каретки для ввода текста (номер строки и номер символа в строке) в заголовок окна (рис. 89).

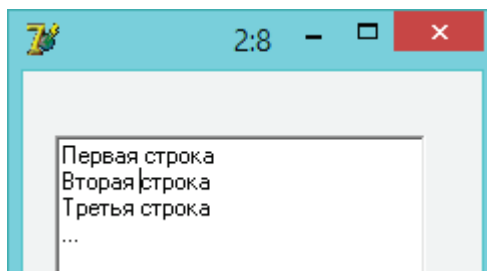


Рисунок 89 – Координаты каретки в заголовке

Это необходимо делать при каждом изменении положения каретки, т.е. при наступлении **OnSelectionChange**, в обработчике которого напишем:

```
procedure TForm1.RichEdit1SelectionChange(Sender: TObject);
begin
    Caption := IntToStr(RichEdit1.CaretPos.Y+1) + ':' +
               IntToStr(RichEdit1.CaretPos.X+1);
end;
```

!!! Также можно обновлять данные о положении каретки не при каждом изменении его положения, а по таймеру, например, каждые 0,2 сек.

Конечно, это был всего лишь пример, и в реальности никто не будет выводить координаты в заголовок окна. Теперь будем выводить информацию о положении каретки в Строчке состояния **StatusBar** (рис. 90).

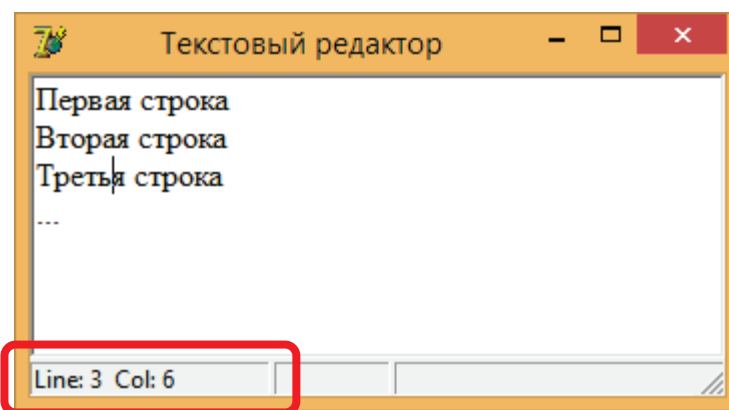


Рисунок 90 – Координаты каретки в панели состояния

Для этого необходимо:

- добавить на форму компонент **TStatusBar** (расположен на вкладке «Win32»), который автоматически выравнивается внизу окна;
- дважды кликнуть по этой панели;
- в появившемся окне (рис. 91) добавить три панели, из которых будет состоять Строчка состояния (см. рис. 90);

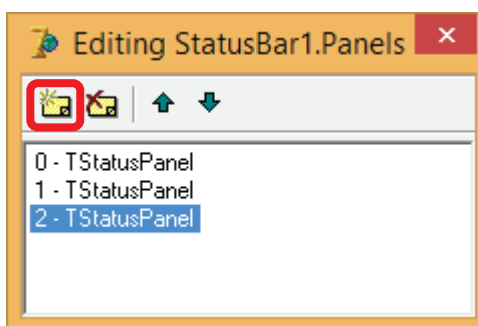


Рисунок 91 – Добавление панелей

- для каждой из этих панелей (кроме последней) необходимо настроить ширину (рис. 92). Также можно написать для них начальный текст.

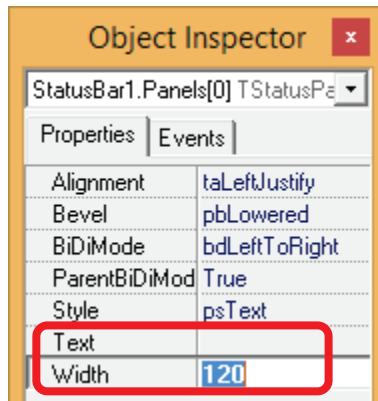


Рисунок 92 – Настройка панелей

Тогда в итоге получим следующий код:

```

{----- Положение каретки -----}
procedure TForm1.RichEdit1SelectionChange(Sender: TObject);
begin
    StatusBar1.Panels[0].Text := 'Line: ' + IntToStr(RichEdit1.CaretPos.Y+1)
    + ' Col: ' + IntToStr(RichEdit1.CaretPos.X+1);
end;

```

!!! Этот пример с **OnSelectionChange** и **StatusBar** стоит запомнить, т.к. он понадобится нам при доработке «Текстового редактора».

2.14. События невидимых компонентов

События для Action-ов

Действия (Action-ы) имеют собственные события и не поддерживают события, перечисленные выше. События **TAction** представлены в таблице 26.

Таблица 26 – События TAction

Событие	Описание
OnExecute: TNotifyEvent;	Выполнение текущего действия. Для всех <u>собственных</u> действий его применение <u>обязательно</u> , иначе они будут недоступны (будут серого цвета). Когда действие связывается с кнопкой или пунктом меню, то OnExecute выполняется вместо их OnClick
OnUpdate: TNotifyEvent;	Позволяет обновлять состояние текущего действия, например, его доступность (Enabled), видимость (Visible), или значение Checked . Не является обязательным
OnHint: THintEvent;	Выполняется перед отображением всплывающей подсказки. Позволяет изменить текст подсказки или запретить ее показ. Применяется редко

Например, будем отображать сообщение «О программе» (рис. 93) при выборе соответствующего действия в меню:

```

{----- О программе -----}
procedure TForm1.About1Execute(Sender: TObject);
begin
  ShowMessage('Текстовый редактор Note' + #13#10 + '© Alexandr Novikov, 2023');
end;

```

!!! Этот пример с *OnExecute* и *ShowMessage* стоит запомнить, т.к. он понадобится нам при доработке «Текстового редактора».

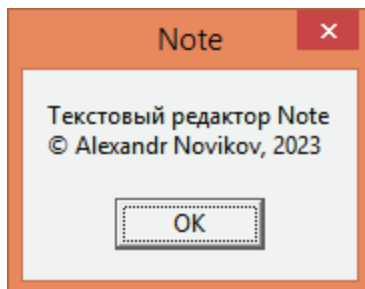


Рисунок 93 – Окно «О Программе»

Другим примером будет Действие (Кнопка) для выделения текста в **RichEdit** красным шрифтом. Если ранее выделенный текст уже был красным, то его необходимо сделать обратно черным:

```

{----- Красный текст -----}
procedure TForm1.RedFontActionExecute(Sender: TObject);
begin
  if RichEdit1.SelAttributes.Color <> clRed
  then RichEdit1.SelAttributes.Color := clRed
  else RichEdit1.SelAttributes.Color := clBlack;
end;

```

При этом мы хотим, чтобы кнопка имела возможность отображаться зажатой, когда выделенный текст уже имеет красный цвет (по аналогии с кнопкой «Жирный» в *Word*):

```

{-----}
procedure TForm1.RedFontActionUpdate(Sender: TObject);
begin
  RedFontAction.Checked := (RichEdit1.SelAttributes.Color = clRed);
end;

```

Диалоговые окна

События для действий, имеющих встроенные диалоговые окна, значительно отличаются (рис. 94) от обычных **TAction**. События **Action-ов** со встроенными диалогами представлены в таблице 27.

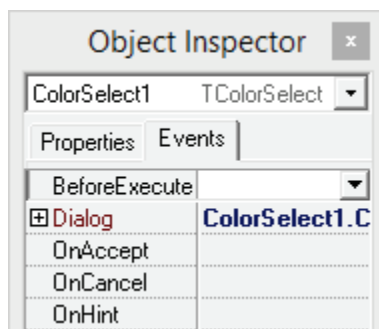


Рисунок 94 – События для действия TColorSelect

Таблица 27 – События Action-ов со встроенными диалогами

Событие	Описание
OnAccept: TNotifyEvent;	Выполняется, когда в диалоговом окне нажата кнопка «Ок». Для почти всех действий с диалогами его применение обязательно, иначе они не будут выполнять ничего полезного, кроме открытия диалога
BeforeExecute: TNotifyEvent; {в нарушение всех правил, пишется без приставки 'On'}	Выполняется непосредственно перед открытием диалогового окна. С его помощью можно задать начальные значения для диалога
OnCancel: TNotifyEvent;	Выполняется, когда в диалоговом окне нажата кнопка «Отмена». Используется редко

Например, для действия выбора цвета (**TColorSelect**) напомним следующий код:

```
{----- Выбрать цвет текста -----}
procedure TForm1.ColorSelect1Accept(Sender: TObject);
begin
    RichEdit1.SelAttributes.Color := ColorSelect1.Dialog.Color;
end;
```

Теперь после нажатия в диалоговом окне кнопки «ОК» выбранный цвет применяется к тексту, который выделен в **RichEdit**. То есть цвет переносится из Диалога в **RichEdit**.

Но в этом случае при открытии Диалога он не будет отображать цвет выделенного в настоящий момент текста (при первом открытии всегда будет черным, а в следующие разы будет отображаться цвет, выбранный в прошлый раз). Т.е. теперь наоборот, требуется переносить цвет из **RichEdit** в Диалог. Тогда необходимо создать также обработчик для события **BeforeExecute**:

```
{-----}
procedure TForm1.ColorSelect1BeforeExecute(Sender: TObject);
begin
    ColorSelect1.Dialog.Color := RichEdit1.SelAttributes.Color;
end;
```

Аналогично для действия выбора шрифта (**TFontEdit**) напомним:

```
{----- Выбрать шрифт -----}
procedure TForm1.FontEdit1Accept(Sender: TObject);
begin
    //Цвет и размер шрифта
    RichEdit1.SelAttributes.Color := FontEdit1.Dialog.Font.Color;
    RichEdit1.SelAttributes.Size := FontEdit1.Dialog.Font.Size;
    //Жирный, курсив, зачеркнутый, подчеркнутый
    RichEdit1.SelAttributes.Style := FontEdit1.Dialog.Font.Style;
    //Имя шрифта и набор символов
    RichEdit1.SelAttributes.Name := FontEdit1.Dialog.Font.Name;
    RichEdit1.SelAttributes.Charset := FontEdit1.Dialog.Font.Charset;
end;
```

```
{-----}
procedure TForm1.FontEdit1BeforeExecute(Sender: TObject);
begin
    //Цвет и размер шрифта
    FontEdit1.Dialog.Font.Color := RichEdit1.SelAttributes.Color;
```

```

FontEdit1.Dialog.Font.Size      := RichEdit1.SelAttributes.Size;
//Жирный, курсив, зачеркнутый, подчеркнутый
FontEdit1.Dialog.Font.Style     := RichEdit1.SelAttributes.Style;
//Имя шрифта и набор символов
FontEdit1.Dialog.Font.Name      := RichEdit1.SelAttributes.Name;
FontEdit1.Dialog.Font.Charset   := RichEdit1.SelAttributes.Charset;
end;

```

Диалоги для работы с файлами

Аналогично для действий сохранения и загрузки файла, необходимо создать обработчик для их события **OnAccept**. Например, если нам требуется сохранять/загрузить содержимое **RichEdit**, то пишется следующий код:

```

//Сохранить в файл
RichEdit1.Lines.SaveToFile (FileSaveAs1.Dialog.FileName);

//Загрузить из файла
RichEdit1.Lines.LoadFromFile (FileOpen1.Dialog.FileName);

```

Если используется **Memo**, то делается точно так же (только **RichEdit** заменяется на **Memo**).

Таймер

Невизуальный компонент **TTimer** расположен на вкладке «**System**». Таймер имеет единственное событие, представленное в таблице 28.

Таблица 28 – События таймера

Событие	Описание
OnTimer: TNotifyEvent;	Выполняется через равные промежутки времени. Промежутки задаются свойством Interval через Инспектор объектов. Время измеряется в <u>миллисекундах</u> . По умолчанию установлено значение 1000 (1 секунда). Таймер можно отключать, устанавливая его свойство Enabled := False . <i>Данный таймер <u>не</u> предназначен для точных замеров времени. Значения интервала меньше 100 задавать нет смысла</i>

Например, будем сдвигать кнопку на **10** пикселей каждые **0,5** секунды. Тогда в обработчике события таймера необходимо написать код:

```
Button1.Left := Button1.Left + 10;
```

Также нужно настроить таймер, установив для него **Interval := 500**.

2.15. События формы

Форма содержит многие из перечисленных ранее событий визуальных компонентов, таких как **OnClick**, **OnDblClick**, **OnContextPopup**, **OnMouseMove**, **OnMouseDown**, **OnMouseUp**, **OnMouseWheel**, **OnMouseWheelDown**, **OnMouseWheelUp**, **OnKeyPress**, **OnKeyDown**, **OnKeyUp**. Но у формы имеется и много собственных событий (табл. 29), которых нет у других компонентов.

Таблица 29 – Основные события формы

Событие	Описание
OnCreate: TNotifyEvent;	Создание формы. Выполняется до того, как окно будет показано в первый раз. Здесь можно задать начальные значения переменным, загрузить исходные данные из файла, создать компоненты (<i>при их динамическом создании</i>) и т.п.
OnCanResize: TCanResizeEvent;	Запрос на изменение размеров окна. Наступает при каждой попытке изменить размеры окна. Здесь можно запретить изменение размера, установив Resize := False ; или задать новые размеры окна NewWidth и NewHeight
OnClose: TCloseEvent;	Уничтожение формы. Выполняется при закрытии программы (<i>для Главной формы, а для остальных может происходить и раньше</i>). Здесь можно сохранить итоговые данные в файл, уничтожить компоненты (<i>которые были созданы динамически</i>) и т.п.
OnCloseQuery: TCloseQueryEvent;	Запрос на закрытие. Наступает при каждой попытке закрыть окно. Здесь, например, можно отобразить диалоговое окно подтверждения закрытия или предложить сохранить файл перед закрытием. Действия в OnCloseQuery выполняются раньше, чем OnClose . Это последняя возможность не дать закрыть программу, для этого необходимо указать значение переменной CanClose := False (по умолчанию CanClose := True)

Например, разместим на форме единственный CheckBox (рис. 95) и напишем следующий код:

```

procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
    if CheckBox1.Checked then CanClose := False;
end;

```

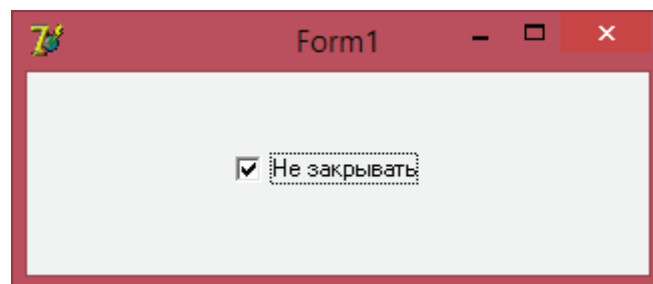


Рисунок 95 – Пример для OnCloseQuery

Данная программа не позволяет закрыть окно до тех пор, пока не будет снята галочка.

2.16. Обзор визуальных компонентов Delphi

Визуальными компонентами являются метки **Label** и поля ввода **Edit** (рис. 96), кнопки **Button** и различные «галочки» (рис. 97), компоненты для отображения изображений, таблиц и графиков (рис. 98), различные компоненты для ввода и/или отображения числовых значений (рис. 99), а также списки (рис. 100), выпадающие списки (рис. 101), панели (рис. 102) и др. *Некоторые компоненты попали сразу на несколько рисунков, т.к. относятся сразу к нескольким группам.*



Рисунок 96 – Метки и поля ввода

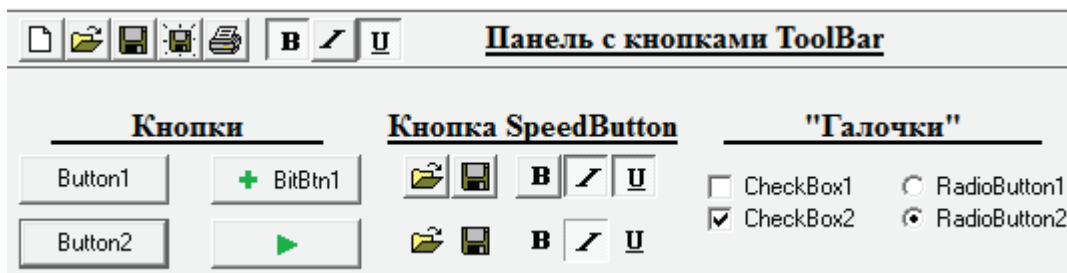


Рисунок 97 – Кнопки и «галочки»

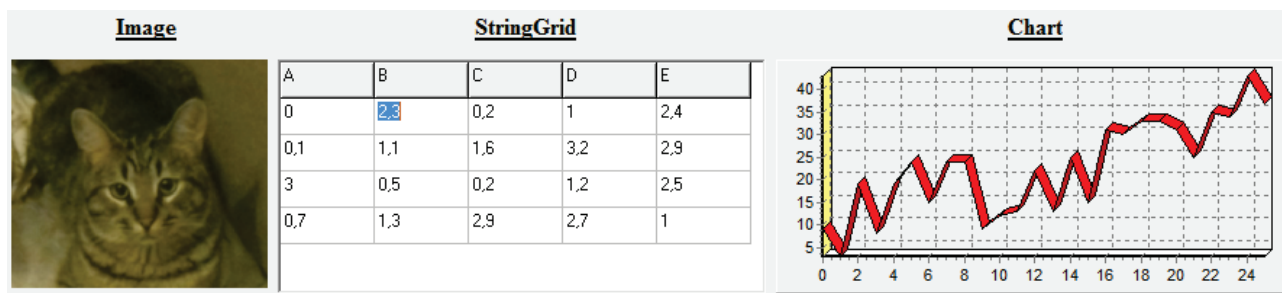


Рисунок 98 – Изображения, таблицы и графики

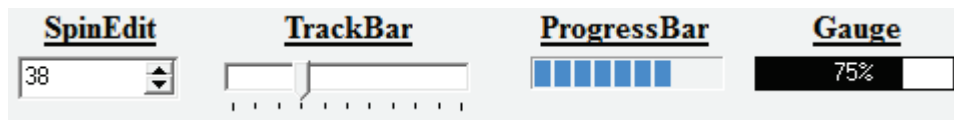


Рисунок 99 – Ввод и отображение числовых значений

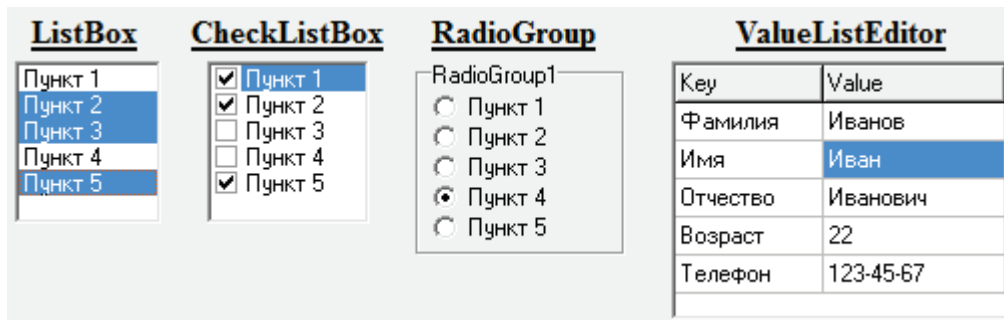


Рисунок 100 – Списки

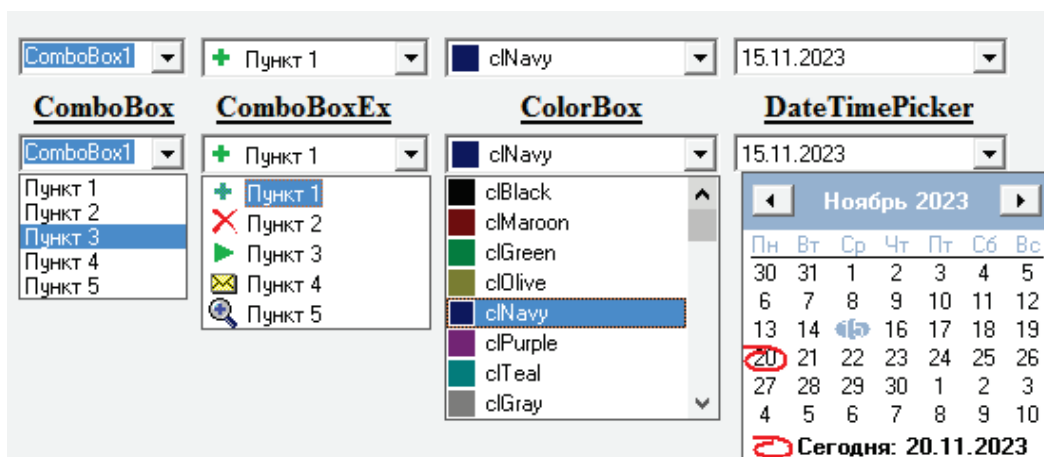


Рисунок 101 – Выпадающие списки

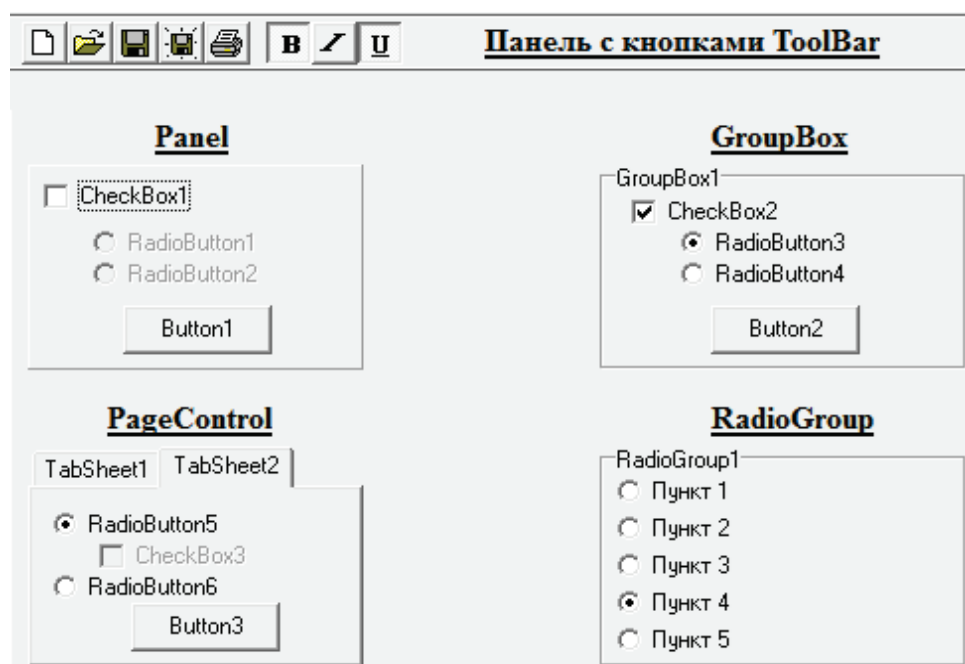




Рисунок 102 – Панели





Основные визуальные компоненты **Delphi** перечислены в таблице 30. Также в таблице приведены основные свойства для каждого из компонентов. При этом здесь не упоминаются общие для всех компонентов свойства такие, как **Enabled**, **Visible**, **Name**, **Left**, **Top**, **Width**, **Height**, **Align**, **PopupMenu**, **Hint**, **ShowHint**, **Cursor** и т.п. Свойства, перечисленные в начале таблицы, могут не повторяться далее для других компонентов, т.к. работают аналогично.




!!! Рекомендуется по мере прочтения таблицы, сразу добавлять указанные компоненты на форму и тестировать их при разных значениях описанных свойств.






Таблица 30 – Основные визуальные компоненты





Компоненты	Описание
Вкладка «Standard»	
 A Label	<p>«Этикетка», «бирка», «пометка» или «метка».</p> <p>Текст «этикетки» задается через свойство Caption. Текст может быть многострочным, для этого необходимо задать свойство WordWrap := True (<i>переносить по словам</i>).</p> <p>Можно изменить цвет текста Font.Color и размер шрифта Font.Size.</p> <p>Можно сделать текст жирным Font.Style := [fsBold] или курсивным Font.Style := [fsItalic]</p>
 Edit	<p>Поле для ввода текста.</p> <p>Основное свойство компонента – Text, типа String.</p> <p>Можно ограничить количество символов, которое возможно ввести, например, MaxLength := 3;</p> <p>Может использоваться для ввода паролей, для этого нужно задать свойство PasswordChar := '*'. Для возврата к отображению всех символов необходимо задать PasswordChar := #0.</p> <p>Можно сделать поле доступным только для чтения (ReadOnly := True), тогда нельзя будет вводить текст, но можно его просматривать. <i>В этом случае Edit станет похож на Label, но в отличие от Label, текст в нем по-прежнему можно будет выделять и копировать (Ctrl+C).</i></p> <p>Можно изменить цвет фона Edit-а, задав для него, например, Color := clBtnFace (<i>что означает серый цвет, такой же, как цвет фона Формы</i>). Или изменить цвет текста Font.Color и размер шрифта Font.Size. Можно выделить текст жирным или курсивом через свойство Font.Style.</p> <p>Можно скрыть рамку вокруг поля, задав свойство BorderStyle := bsNone. <i>Тогда он станет совсем похож на Label.</i></p>





Компоненты	Описание
	<p>Можно сделать отображение всех букв большими (ПРОПИСНЫМИ) CharCase := <code>ecUpperCase</code>, или маленькими (строчными) CharCase := <code>ecLowerCase</code>. К полю Edit, как и к большинству других полей ввода, уже привязано системное всплывающее меню, которое появляется при нажатии правой клавиши мышки. Оно содержит пункты «Копировать», «Вставить», «Выделить все» и т.д. Можно вместо него привязать свое всплывающее меню через свойство PopupMenu. Для полного отключения всплывающего меню необходимо привязать пустой компонент TPopupMenu</p>
 Memo	<p>Многострочный текст. Основное свойство компонента – Lines, типа TStrings (т.е. многострочный текст). Для этого компонента доступны такие свойства как: WordWrap – переносить по словам; ReadOnly – только для чтения; Color, Font.Color, Font.Size – цвет фона, цвет текста и размер шрифта; BorderStyle – рамка вокруг компонента; MaxLength – максимальное количество вводимых символов (причем каждые Enter считается как два символа); ScrollBars – включение и выключение полос прокрутки (вертикальной – справа, и горизонтальной – снизу).</p>
 Button	<p>Кнопка. Надпись на кнопке задается через свойство Caption. Ее можно сделать многострочной через свойство WordWrap. Для кнопки можно изменять размер шрифта Font.Size или сделать текст жирным или курсивом через свойство Font.Style. Но цвет фона Color или цвет текста Font.Color изменить нельзя. У кнопок есть особое свойство Default типа Boolean. Если его включить, то кнопка будет выделена более темной рамкой и станет «кнопкой по умолчанию». Это означает, что при нажатии клавиши Enter на любом поле ввода (таком как Edit), сработает эта кнопка (точнее ее событие OnClick). Аналогичное произойдет при нажатии Enter для списка и многих других компонентов. Но это не работает для многострочных полей Memo и RichEdit, т.к. у них нажатие Enter означает переход на новую строку. Только для одной кнопки можно задать Default := True</p>







Компоненты	Описание
 CheckBox	<p>«Галочка», «Флажок», «Чекбокс».</p> <p>Основное свойство компонента – Checked, типа Boolean. Надпись на компоненте задается через свойство Caption. Ее можно сделать многострочной через свойство WordWrap.</p>
 RadioButton	<p>«Радиокнопка».</p> <p>Основное свойство компонента – Checked, типа Boolean. Надпись задается через Caption, для которого можно задать WordWrap.</p> <p>Радиокнопка очень похожа на CheckBox, но предназначена только для работы <u>в группе</u> из нескольких RadioButton-ов (<i>не менее двух</i>). Все радиокнопки, помещенные на форму, автоматически объединяются в одну группу. Для того чтобы образовать отдельную группу радиокнопок, необходимо использовать панели (Panel, GroupBox и т.п.)</p>
 ListBox	<p>Список.</p> <p>Список вводится как многострочный текст через свойство Items: TStrings. Каждая строка исходного текста будет отдельным элементом списка.</p> <p>По списку можно перемещаться клавишами «Вверх» и «Вниз» («↑», «↓») на клавиатуре.</p> <p>Если задать свойство MultiSelect := True, то можно выделять не один, а сразу несколько пунктов списка. Выделять несколько пунктов списка <u>подряд</u> можно удерживая кнопку Shift. Чтобы выделить пункты <u>не</u> подряд, необходимо удерживать Ctrl</p>
 ComboBox	<p>Выпадающий список (он же «Поле со списком»).</p> <p>Список вводится как многострочный текст через свойство Items: TStrings.</p> <p>Т.к. ComboBox является не только списком, но и полем для ввода (как Edit), то его основным свойством, как и у других полей, является Text. В это поле можно вводить любой текст, а не только указанный в списке. ComboBox обладает и некоторыми другими свойствами полей Edit, такими как: MaxLength, или CharCase.</p> <p>Если задать свойство AutoDropDown := True, то при вводе текста с клавиатуры, список будет автоматически раскрываться и выбирать первый из пунктов, подходящий под введенный текст. По списку можно перемещаться клавишами «Вверх» и «Вниз» («↑», «↓») на клавиатуре.</p> <p>Свойство компонента ItemIndex указывает номер выбранного пункта списка (начиная с нуля). Если ни</p>





Компоненты	Описание
	<p>один пункт не выбран, то ItemIndex = -1.</p> <p>Можно задать свойство Style := csDropDownList, тогда можно будет выбирать только значения из списка, и нельзя вводить произвольный текст. В этом случае основным свойством может выступать уже ItemIndex, а Text можно вообще не использовать.</p> <p>Ограничить количество строк, отображаемых в выпадающем списке, можно при помощи свойства DropDownCount</p>
 Panel	<p>Панель.</p> <p>Все панели предназначены для <u>визуального</u> объединения компонентов в одну группу. В случае с Радиокнопками это объединение еще и <u>функциональное</u>.</p> <p>Панели являются «<u>Контейнерами</u>», т.е. в них можно разместить любые другие визуальные компоненты (в т.ч. другие панели). Форма TForm также является контейнером.</p> <p>Можно отключить рамку вокруг компонента Panel, задав свойство BevelOuter := bvNone. Другими свойствами, отвечающими за различные варианты отображения рамки, являются: BevelInner, BevelWidth и BorderStyle. Для этой панели обязательно нужно задавать Caption:=' ', т.к. надпись посередине панели совершенно не нужна</p>
 GroupBox	<p>Панель с заголовком и рамкой.</p> <p>Используется как Контейнер. Заголовок задается через Caption</p>
 RadioGroup	<p>Группа радиокнопок, Панель с радиокнопками.</p> <p>Внешне совпадает с GroupBox (т.е. с панелью с заголовком и рамкой), но, на самом деле, <u>не является контейнером</u> (в нее нельзя помещать другие компоненты).</p> <p>Является видом Списка (как ListBox или ComboBox) и имеет свойство Items: TStrings с многострочным текстом, каждая строка которого добавляет отдельную радиокнопку на данную «панель».</p> <p>Можно задать количество колонок при помощи свойства Columns.</p> <p>Основное свойство компонента – ItemIndex, указывающее номер выбранной радиокнопки (начиная с нуля). Если ни одна радиокнопка не выбрана, то ItemIndex = -1. <i>Стоит напомнить, что только одна радиокнопка в группе может быть выбрана</i></p>



Компоненты	Описание
Вкладка «Additional»	
 BitBtn	<p>Вариант кнопки с изображением. Изображение задается через свойство Glyph. В остальном данная кнопка аналогична обычной кнопке TButton, в т.ч. для нее можно задать свойство Default</p>
 SpeedButton	<p>Вариант кнопки с изображением. Изображение также задается через свойство Glyph, но данная кнопка сильно отличается от обычной кнопки TButton, в т.ч. у нее <u>нет</u> свойства Default. Зато она позволяет зажать кнопку в нижнем положении (свойство Down), а также объединять кнопки в группы при помощи свойства GroupIndex. Если GroupIndex = 0, то группировка отключена и не удастся зажать кнопку. В группе с одинаковым номером может быть зажата только одна кнопка. Если мы хотим, чтобы все кнопки в группе могли быть отжаты (т.е. были в верхнем положении), то необходимо установить AllowAllUp := True. У этой кнопки все еще имеется свойство Caption, поэтому мы, как и прежде, можем задать надпись на кнопке. Свойство Flat := True – позволяет скрыть рамку вокруг кнопки</p>
 MaskEdit	<p>Поле ввода с маской. Предназначено для ввода номеров телефонов, даты и времени, почтовых индексов и др. значений, для которых заранее задан определенных формат записи. Маска вводится через свойство EditMask. Если дважды щелкнуть по этому значению (или нажать в нем кнопку с многоточием «...»), то откроется Мастер, в котором можно выбрать один из готовых шаблонов. <i>К сожалению, имеющиеся варианты плохо подходят для России, поэтому их придется немного подправить под себя</i></p>
 StringGrid	<p>Таблица. Количество строк и столбцов задается свойствами RowCount и ColCount. По умолчанию для таблицы отключен ввод значений в ячейки с клавиатуры. Для его включения необходимо в свойстве Options установить goEditing := True</p>
 Image	<p>Изображение. Основное свойство – Picture. В него можно загрузить изображение в формате BMP, JPEG и др. Если размеры загруженного изображения отличаются от размеров самого Image, то необходимо установить</p>

Компоненты	Описание
	свойство Proportional := True, тогда изображение будет промасштабировано под размер компонента, но с сохранением пропорций исходного изображения. <i>Если пропорции не важны, то вместо Proportional можно установить свойство Stretch := True</i>
 Bevel	<p>Рисует рамки и линии.</p> <p>Свойство Shape позволяет выбрать тип рамки или тип линии (вертикальная или горизонтальная)</p> <p>Не является панелью и не может быть контейнером для других компонентов. У этого компонента полностью отсутствуют какие-либо события (нет даже OnClick). Но для данной области можно задать свои Cursor и Hint</p>
 CheckListBox	<p>Список «галочек».</p> <p>Список вводится как многострочный текст через свойство Items: TStrings. Каждая строка исходного текста будет отдельным элементом списка.</p> <p>Список будет автоматически отсортирован по возрастанию, если включить Sorted := True.</p> <p>По списку можно перемещаться клавишами «Вверх» и «Вниз» («↑», «↓») на клавиатуре, а галочки включать и выключать нажатием клавиши «Пробел».</p> <p>Каждая галочка включается независимо, можно включить любое их количество (в т.ч. можно выключить все).</p> <p>Можно задать количество колонок при помощи свойства Columns.</p> <p>Можно скрыть рамку вокруг поля, задав свойство BorderStyle := bsNone, или изменить цвет фона Color (например, на тот же цвет, что имеет окно)</p>
 Splitter	<p>Разделитель.</p> <p>Позволяет изменять размеры панелей и компонентов пользователем во время работы программы (в Run-time).</p> <p>Применяется к компонентам, для которых установлено значение Align. Для самого Splitter-а, также нужно задать соответствующий Align.</p> <p>Может быть как горизонтальным, так и вертикальным</p>
 ValueListEditor	<p>Список значений.</p> <p>Таблица из двух колонок «Key» («Ключ», «Имя») и «Value» («Значение»).</p> <p>Основное свойство – Strings типа TStrings. Через него задается список «Ключей» и их «Значения» по умолчанию.</p> <p>Например, можно создать таблицу со строками: Фамилия, Имя, Отчество и Возраст, а пользователь впишет свои значения.</p>

Компоненты	Описание
	<p>Через свойство TitleCaptions можно изменить заголовки колонок в шапке таблицы (изначально это «Key» и «Value»).</p> <p>ValueListEditor имеет множество настроек, осуществляемых через свойства Options, KeyOptions и DisplayOptions.</p> <p>Во время работы программы, пользователь может изменять только «Значения», но не имена «Ключей». Кроме того, пользователь не может добавлять новые строки или удалять существующие. Но через свойство KeyOptions можно разрешить изменение, добавление и удаление «Ключей». Удаление строки («Ключа») в этом случае производится нажатием Ctrl+Del. Для добавления новой строки необходимо переместиться клавишей клавиатуры «Вниз» («↓») ниже последней строки, чтобы появилась новая строка</p>
 LabeledEdit	<p>Вариант поля Edit со встроенной надписью Label. Текст задается, как и обычно для Edit-a, через свойство Text, а надпись через EditLabel.Caption. Подпись может располагаться сверху, снизу, слева или справа, что задается свойством LabelPosition</p>
 ColorBox	<p>Выпадающий список с цветами.</p> <p>Главное свойство – Selected, показывающее, какой цвет выбран в списке.</p> <p>Настроить список доступных для выбора цветов можно через свойство Style. В нем рекомендуется отключать системные цвета cbSystemColors, а также включить cbCustomColor, который позволит выбирать не только фиксированные цвета из списка, но и вызвать Диалог для выбора произвольного цвета</p>
 Chart	<p>График</p>
Вкладка «Win32»	
 PageControl	<p>Многостраничная панель.</p> <p>Позволяет экономить пространство окна, распределяя информацию по вкладкам. Часто используются для окон с настройками. Каждая вкладка (страница), фактически является отдельной панелью типа TTabSheet (<i>и отдельным независимым контейнером для размещения других компонентов</i>).</p> <p>Для добавления новой вкладки необходимо нажать на PageControl правой клавишей мышки и выбрать пункт меню «New Page».</p>

Компоненты	Описание
	Надпись на вкладке задается через свойство Caption . Вкладки могут иметь не только надписи, но и иконки, выбираемые через ImageIndex . При этом PageControl должен быть связан с ImageList (в который заранее нужно загрузить изображения)
 RichEdit	<p>Многострочный текст с форматированием. Позволяет выделить текст жирным или курсивом, изменить цвет или размер шрифта и т.п. Основное свойство компонента – Lines, типа TStrings. <i>На этапе проектирования можно указать только текст без форматирования! Форматирование можно применить во время работы программы. Можно вставлять форматированный текст из Word-а или копировать его в Word. Также форматированный текст можно загрузить из файла в формате *.rtf.</i></p> <p>В остальном практически полностью совпадает с Memo</p>
 TrackBar	<p>«Ползунок».</p> <p>Может перемещаться как мышкой, так и клавишами клавиатуры «Влево», «Вправо», «Вверх», «Вниз» («←», «→», «↑», «↓»).</p> <p>При помощи свойства Orientation можно сделать его горизонтальным или вертикальным.</p> <p>Основное свойство компонента – Position типа Integer. Также можно задать значения границ Min и Max (по умолчанию от 0 до 10)</p>
 ProgressBar	<p>Отображение процента выполнения.</p> <p>При помощи свойства Orientation можно сделать его горизонтальным или вертикальным.</p> <p>Основное свойство компонента – Position типа Integer. Также можно задать значения границ Min и Max (по умолчанию от 0 до 100)</p>
 MonthCalendar  DateTimePicker	<p>Календарь и выпадающий «список» с календарем</p>
 TreeView	<p>Дерево.</p> <p>Позволяет отображать иерархические данные в виде дерева (например, такие, как структура каталогов на диске ПК).</p> <p>Отдельные ветви дерева можно сворачивать и разворачивать</p>

Компоненты	Описание
 StatusBar	Строка состояния (она же Панель состояния), расположенная внизу окна
 ToolBar	<p>Панель с кнопками, обычно расположенная сверху. Для добавления новой кнопки необходимо нажать на ToolBar правой клавишей мышки и выбрать пункт меню «New Button». Кроме того, можно добавить разделители между кнопками, выбрав «New Separator».</p> <p>Изображения на кнопках задаются через свойство ImageIndex. При этом ToolBar должен быть связан с ImageList (в который заранее нужно загрузить изображения).</p> <p>Свойство Flat := True – позволяет скрыть рамки вокруг кнопок (как это было у SpeedButton-ов).</p> <p>Если для кнопки задать Style := tbsCheck, то эту кнопку можно будет переключать между нижним (зажатым) и верхним (отжатым) положением. Кнопка будет находится в нижнем положении при Down := True, а в верхнем положении при Down := False.</p> <p>Для объединения кнопок в группы необходимо задать для них Grouped := True. Все кнопки (с включенным Grouped), идущие подряд, образуют единую группу. Для разделения кнопок на несколько групп, между ними должна располагаться кнопка с отключенной группировкой, разделитель (Separator), или любой иной компонент (выпадающий список, чекбокс, радиокнопка и т.п.). В группе может быть одновременно зажата только одна кнопка. Если мы хотим, чтобы все кнопки в группе могли быть отжаты (т.е. быть в верхнем положении), то необходимо установить AllowAllUp := True.</p> <p>ToolBar является «Контейнером», поэтому на эту панель можно добавить не только стандартные кнопки и разделители, но и любые другие визуальные компоненты (например, выпадающие списки, чекбоксы и радиокнопки)</p>
 ComboBoxEx	Вариант выпадающего списка, позволяющий добавлять к пунктам меню иконки из ImageList
Вкладка «Samples»	
 Gauge	<p>Отображение процента выполнения.</p> <p>При помощи свойства Kind можно сделать его горизонтальным, вертикальным, круглым или полукруглым.</p> <p>Основное свойство компонента – Progress типа Integer.</p> <p>Также можно задать значения границ MinValue и</p>

Компоненты	Описание
	MaxValue (по умолчанию от 0 до 100)
 ColorGrid	Выбор цвета. Позволяет выбирать только из 16-ти фиксированных цветов. При этом можно выбирать сразу два цвета, один правой кнопкой мышки, другой – левой
 SpinEdit	Поле ввода числовых значений. Основное свойство компонента – Value , типа Integer . Можно ограничить диапазон вводимых чисел при помощи свойств MaxValue и MinValue . Содержит дополнительные кнопки увеличения и уменьшения значения. Используя свойство Increment можно задать шаг изменения значений при помощи этих кнопок

Пример бурной деятельности

Необходимо каждую секунду генерировать случайное число, отображаемое в **Edit**, а также увеличивать значение **ProgressBar** на **2%**. При достижении **100%** остановить генерацию. Внешний вид программы представлен на рис. 103. *Вместо **ProgressBar** можно также использовать компонент **Gauge**.*

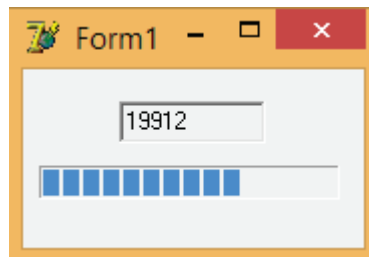


Рисунок 103 – Пример бурной деятельности

Код для данной программы будет следующим:

```

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Randomize;
  Edit1.Text := IntToStr(Random(100000));

  ProgressBar1.Position := ProgressBar1.Position + 2;
  if ProgressBar1.Position >= 100 then Timer1.Enabled := False;
end;

```

2.17. Графический интерфейс пользователя

GUI (**Graphical User Interface**, Графический интерфейс пользователя) – система средств для взаимодействия пользователя с электронными устройствами, основанная на представлении всех доступных пользователю системных объектов в виде графических компонентов экрана (окон, значков, меню, кнопок, списков и т. п.).

GUI является стандартной составляющей большинства современных операционных систем, таких как: Microsoft Windows, Mac OS, Linux, Android, iOS и др.

GUI-конструктор (GUI-редактор, Конструктор графического пользовательского интерфейса) является инструментарием разработки программного обеспечения, который упрощает создание графического интерфейса пользователя (**GUI**), позволяя разработчику упорядочить элементы интерфейса, используя **drag-and-drop** («тащи-и-бросай») при помощи мышки. Без **GUI-конструктора** графический интерфейс пользователя приходится создавать вручную, с указанием параметров каждого элемента интерфейса в исходном коде без визуальной обратной связи (до запуска программы). Пользовательские интерфейсы обычно программируются с помощью **событийно-ориентированной** архитектуры, поэтому **GUI-конструкторы** также упрощают создание кода, управляемого событиями.

Delphi (и **C++ Builder**) имеет в своем составе **GUI-редактор**, в котором и создаются окна будущей программы.

В противоположность Графическому интерфейсу существует также «**Интерфейс командной строки**». *Delphi* позволяет разрабатывать и такие приложения, но все же основное его назначение – создание программ с Графическим интерфейсом.

2.18. Свойства и методы компонентов

Свойства компонентов

Ранее уже были рассмотрены основные свойства компонентов, как визуальных (см. раздел 2.4), так и некоторых невизуальных (см., например, раздел 2.8). Свойства можно изменять при проектировании программы (в **Design-time**) через «Инспектор объектов». События компонентов также являются одним из видов свойств, и их также можно задавать через «Инспектор объектов» (но необходимо перейти на вкладку «**Events**»).

Все свойства, которые можно изменять через «Инспектор объектов», можно также изменять и из кода (т.е. в **Run-time**). Например:

```
Edit1.Text := 'Привет!'; //Изменяем текст из кода
```

Их мы будем называть «визуальными» свойствами.

Кроме того, у компонентов существуют также свойства, которые нельзя задавать через «Инспектор объектов» (в **Design-time**). Их можно изменять только из кода программы (в **Run-time**). Например:

```
Edit1.SelStart := 3; //Изменяем положение каретки
```

Их мы будем называть «невизуальными» свойствами.

Методы компонентов

Описанные выше Свойства компонентов можно рассматривать как специальный вид «переменных», но не обычных переменных, объявленных как **var**, а как переменных, принадлежащих компоненту.

Кроме Свойств у компонентов существуют еще и **Методы**.

Методы – это процедуры и функции, но не любые, а только принадлежащие к конкретному компоненту. Эти процедуры и функции не используются в **Design-time** (и никогда не отображаются «Инспектор объектов»), их можно использовать только в коде программы, например:

```
Edit1.Clear; //Процедура, очищающая содержимое поля Edit
```

2.19. Свойства и методы полей Edit, Memo и RichEdit

Основные невидимые свойства и методы полей для ввода текста (**Edit**, **Memo**, **RichEdit**) представлены в таблице 31. В данном разделе мы не будем повторно рассматривать визуальные свойства компонентов, которые отображаются в Инспекторе объектов.

Таблица 31 – Основные невидимые свойства и методы полей ввода

Свойства и методы	Описание
procedure Clear ; procedure ClearSelection ;	Очищает все содержимое поля или очищает выделенный текст
procedure CopyToClipboard ; procedure CutToClipboard ; procedure PasteFromClipboard ;	Копирует или вырезает в буфер обмена выделенный текст. Или вставляет текст из буфера обмена. Вставка происходит в ту позицию, где находится каретка. <i>Для этих и некоторых других методов в Delphi уже созданы стандартные Action-ы. Они используют эти методы, поэтому пользователю их уже можно не писать. Но стоит помнить, что стандартные Action-ы применяются не к конкретному компоненту, а к тому компоненту, который находится в фокусе</i>
procedure SelectAll ;	Выделяет весь текст в поле
procedure Undo ;	Отменяет последнее действие. <i>Повторное применение отменяет отмену, т.е. возвращает текст, который был до применения первой отмены</i>
SelStart : Integer; SelLength : Integer; SelText : String;	Положение каретки (или начало выделения текста), длина выделенного текста (количество выделенных символов) и текст выделенного фрагмента. <i>Стоит обратить внимание, что Enter (т.е. переход на новую строку) занимает сразу 2 символа</i>
function Focused : Boolean; procedure SetFocus ;	Focused – возвращает True, если фокус находится на данном компоненте; SetFocus – устанавливает фокус на данный компонент

Свойства и методы	Описание
Modified: Boolean;	<p>«Модифицирован», «Изменен».</p> <p>Автоматически устанавливается в True при вводе текста пользователем.</p> <p><i>При изменении текста из кода, Modified не изменяется для Edit и Memo. Но для RichEdit Modified изменяется и из кода (например при вводе RichEdit1.Lines.Text := '123').</i></p> <p>Сбрасывается в False только вручную (из кода). Может использоваться для того, чтобы определить, нужно ли сохранять текст при закрытии программы (был ли документ изменен)</p>

Будем отображать состояние **Modified** в панели состояния. Обновлять значение будем по таймеру:

```

{----- Обновление по таймеру -----}
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if RichEdit1.Modified then StatusBar1.Panels[1].Text := 'Modified'
    else StatusBar1.Panels[1].Text := '';
end;

```

При этом для таймера необходимо задать время **0,1-0,2** секунды.

Значение **Modified** нужно сбрасывать вручную. Тогда для сохранения и загрузки файла правильно будет писать:

```

//Сохранить в файл
RichEdit1.Lines.SaveToFile(FileSaveAs1.Dialog.FileName);
RichEdit1.Modified := False;

//Загрузить из файла
RichEdit1.Lines.LoadFromFile(FileOpen1.Dialog.FileName);
RichEdit1.Modified := False;

```

Для создания нового документа используется следующий код:

```

//Создание нового (чистого) документа
RichEdit1.Clear;
RichEdit1.Modified := False;

```

!!! Эти примеры с *Modified* и *Clear* стоит запомнить, т.к. они понадобятся нам при доработке «Текстового редактора».

Многострочные поля

Многострочные поля **Memo** и **RichEdit** включают все перечисленные выше свойства и методы, а также содержат некоторые дополнительные (табл. 32), которых нет у обычных полей **Edit**.

Таблица 32 – Основные невидимые свойства многострочных полей

Свойства	Описание
CaretPos : TPoint; <i>где TPoint</i> = record X: Longint; Y: Longint; end;	Положение каретки. Где CaretPos.Y – номер строки (начиная с нуля); CaretPos.X – номер символа в этой строке (начиная с нуля)
Lines : TStrings;	Многострочный текст. Содержит следующие собственные свойства и методы: Lines.Text – текст; Lines.Count – количество строк текста; Lines.Add – добавляет новую строку в конец; Lines.LoadFromFile, Lines.SaveToFile – загружает текст из файла, или сохраняет текст в файл. Для Memo это обычный текст в формате *.txt. Для RichEdit это форматированный текст *.rtf; Lines.Delete, Lines.Insert, Lines.Move, Lines.Exchange – удаление, добавление и перемещение строк; и др.

Например, создадим новый текстовый документ, состоящий из трех строк. Если в поле **Мемо** раньше был другой текст, то он будет потерян:

```

Memo1.Clear;
Memo1.Lines.Add('Строка 1');
Memo1.Lines.Add('Строка 2');
Memo1.Lines.Add('Строка 3');

```

Поля RichEdit

Для форматированного текста **RichEdit** имеются дополнительные свойства и методы (табл. 33).

Таблица 33 – Основные невидимые свойства и методы RichEdit

Свойства и методы	Описание
SelAttributes : TTextAttributes;	Задает формат для выделенного текста. Например: RichEdit1.SelAttributes.Color := clRed; RichEdit1.SelAttributes.Style := [fsBold];
Paragraph : TParaAttributes;	Задает формат для параграфа, на котором находится каретка. <i>Можно задавать формат сразу для нескольких параграфов, для этого необходимо выделить текст в этих параграфах.</i> Например: RichEdit1.Paragraph.Alignment := taCenter; RichEdit1.Paragraph.FirstIndent := 10;

Свойства и методы	Описание
procedure Print (Caption: String);	Выводит документ на печать. Печать осуществляется на принтере, который выбран через PrintDialog (или через Action, содержащий этот диалог). Caption указывает название документа, которое будет отображаться в «Очереди печати». При печати на виртуальный PDF-принтер, это будет еще и имя сохраняемого файла

Для печати форматированного текста, используется следующий код:

```

{----- Печать документа -----}
procedure TForm1.PrintDlg1Accept(Sender: TObject);
var Caption: String;
begin
    Caption := 'Новый документ';
    RichEdit1.Print(Caption);
end;

```

!!! Этот пример с *Print* стоит запомнить, т.к. он (как и следующие примеры с мышкой) уже скоро понадобится нам при разработке «Графического редактора».

Для типа **TTextAttributes** и, соответственно, для свойства **SelAttributes**, имеются следующие собственные свойства (табл. 34).

Таблица 34 – Основные свойства TTextAttributes

Свойства и методы	Описание
Size : Integer;	Размер шрифта
Color : TColor;	Цвет текста
Name : TFontName;	Имя шрифта
Style : TFontStyles;	Позволяет сделать текст жирным, курсивным, подчеркнутым или зачеркнутым. Например: RichEdit1.SelAttributes.Style := [fsBold]; или: RichEdit1.SelAttributes.Style := [fsBold, fsItalic];

Для типа **TParaAttributes** и, соответственно, для свойства **Paragraph**, имеются следующие собственные свойства (табл. 35).

Таблица 35 – Основные свойства TParaAttributes

Свойства и методы	Описание
Alignment : TAlignment; где TAlignment = (taLeftJustify, taRightJustify, taCenter);	Выравнивание текста по левому краю, по правому краю или по центру

Свойства и методы	Описание
Numbering: TNumberingStyle; где TNumberingStyle = (nsNone, nsBullet)	Вставка маркеров списка
FirstIndent: Longint;	Отступ красной строки (в пикселях)
LeftIndent: Longint; RightIndent: Longint;	Отступы от левого и правого краев (в пикселях)

2.20. Диалоговые окна Application.MessageBox

Ранее мы уже рассмотрели процедуру **ShowMessage**, которая отображала диалоговое окно. Но у нее был всего один параметр – текст, отображаемый в окне. Для нее нельзя было изменить название окна, и это окно содержало только единственную кнопку «ОК».

Больше возможностей можно получить, используя функцию **Application.MessageBox**, объявленную как:

```
function MessageBox(Text, Caption: PChar; Flags: Longint): Integer;
```

где **Text** и **Caption** – текст окна и заголовок окна;





Flags – сумма флагов с различными дополнительными параметрами.

Флаги отвечают за количество кнопок (табл. 36) и изображение в окне (табл. 37).

Таблица 36 – Кнопки диалогового окна

Значение	Кнопки
MB_OK	Единственная кнопка «ОК»
MB_OKCANCEL	«ОК», «Отмена»
MB_YESNO	«Да», «Нет»
MB_YESNOCANCEL	«Да», «Нет», «Отмена»
MB_RETRYCANCEL	«Повтор», «Отмена»
MB_ABORTRETRYIGNORE	«Прервать», «Повтор», «Пропустить»

Таблица 37 – Изображения диалогового окна

Значение	Описание
 MB_ICONWARNING	Восклицательный знак (замечание, предупреждение)
 MB_ICONINFORMATION	Буква «i» в круге (информация)
 MB_ICONQUESTION	Знак вопроса (вопрос, ожидание ответа)
 MB_ICONERROR	Крест на красном фоне (ошибка, стоп, запрет)

Изображение не является обязательным для диалогового окна. Его можно вообще не указывать.

Кроме того, при помощи флагов можно задать кнопку по умолчанию, которая будет выбрана при открытии окна диалога (табл. 38). Для использования варианта, выбранного по умолчанию, в окне достаточно нажать клавишу «**Enter**».

Таблица 38 – Кнопки по умолчанию

Значение	Кнопка
MB_DEFBUTTON1	Выбрана первая кнопка (значение по умолчанию)
MB_DEFBUTTON2	Выбрана вторая кнопка
MB_DEFBUTTON3	Выбрана третья кнопка

В зависимости от того, какая кнопка была нажата, функция возвращает один из результатов, представленных в таблице 39.

Таблица 39 – Возвращаемое значение

Значение	Кнопка
mrOk	«ОК»
mrCancel	«Отмена»
mrYes	«Да»
mrNo	«Нет»
mrRetry	«Повтор»
mrAbort	«Прервать»
mrIgnore	«Пропустить»

!!! Стоит обратить внимание, что строки в данном случае имеют тип **PChar**, а не **String**. Для константных значений, которые сразу указываются в кавычках внутри функции, это не имеет значения. Но если мы захотим использовать некоторую переменную **S** типа **String** внутри функции, то ее нужно будет записывать как **PChar(S)**, т.е. использовать преобразование типов.

Например, для отображения окна (рис. 104) с предложением заменить файл, необходимо выполнить следующий код:

```

if Application.MessageBox('Заменить файл?', 'Файл существует',
    MB_YESNO + MB_ICONQUESTION +
    MB_DEFBUTTON2) = mrYes then
begin
    //Выполнить сохранение файла
end;

```

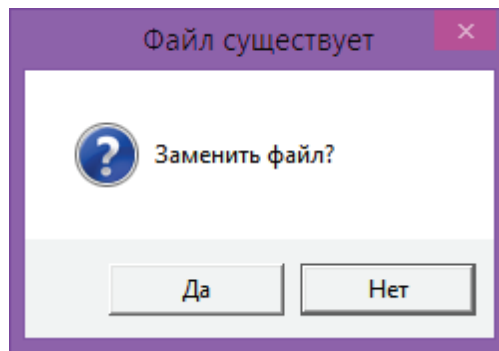


Рисунок 104 – Диалог замены файла

Или, например, для отображения окна (рис.105) с предложением сохранить документ при закрытии программы, необходимо выполнить следующий код:

```
R := Application.MessageBox('Сохранить изменения в документе?',
    'Документ изменен',
    MB_YESNOCANCEL + MB_ICONQUESTION);
```

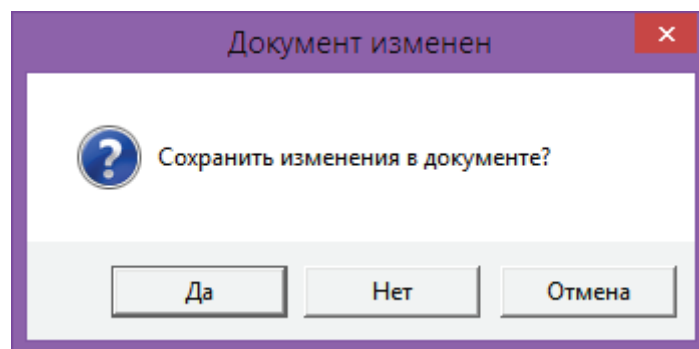


Рисунок 105 – Диалог сохранения изменений

Данное окно содержит три кнопки «Да», «Нет», «Отмена» (MB_YESNOCANCEL). Результат с выбранной пользователем кнопкой сохраняется в переменную **R**, типа **Integer**.

Пример использования данного окна, может быть следующим:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var R: Integer;
begin
    //Если файл был изменен
    if RichEdit1.Modified then
    begin
        //Отображаем диалоговое окно
        R := Application.MessageBox('Сохранить изменения в документе',
            'Документ изменен',
            MB_YESNOCANCEL + MB_ICONQUESTION);

        //Отменяем закрытие программы
        if R = mrCancel then CanClose := False;
        //Сохраняем файл
        if R = mrYes then FileSave1.Execute;
    end;
end;
```

!!! Этот пример с *Application.MessageBox* стоит запомнить, т.к. он понадобится нам при доработке «Текстового редактора».

Учебное издание

Новиков Александр Игоревич

**Алгоритмизация и программирование
Выполнение контрольных работ для студентов
первого курса заочной формы обучения**

Учебно-методическое пособие

Редактор и корректор Д. А. Романова
Техн. редактор Д. А. Романова

Темплан 2024 г., поз. 5070/24

Подписано к печати 10.04.2024.	Формат 60x84/16.	Бумага тип № 1.
Печать офсетная.	Печ.л. 7,4.	Уч.-изд. л. 7,4.
Тираж 30 экз.	Изд. № 5070/24.	Цена «С». Заказ №

Ризограф Высшей школы технологии и энергетики СПбГУПТД,
198095, Санкт-Петербург, ул. Ивана Черных, 4.