***ЛАБОРАТОРНОЕ ЗАНЯТИЕ № 1-2.***

*Тема:* Применение информационно-коммуникационных технологий с учетом основных требований информационной безопасности: Организация работы с модулями в среде программирования Паскаль

СОЗДАНИЕ ОРИГИНАЛЬНЫХ МОДУЛЕЙ

Структура модуля имеет следующий вид:

```
UNIT <Имя модуля>;

INTERFACE                  // Раздел объявлений процедур и
                           // функций модуля
<Заголовки процедур и функций модуля>

IMPLEMENTATION             // Раздел описания объявленных
                           // в разделе INTERFACE
                           // процедур и функций модуля
<Описания процедур и функций модуля>
End.                       // окончание модуля.
```

Рассмотрим примеры реализации модулей.

ПРИМЕР №1. Прорисовка прямоугольника на алфавитно-цифровом экране.
ПРИМЕР №2. Моделирование работы стека.

**ПРИМЕР №1**. Прорисовка прямоугольника на алфавитно-цифровом экране.

Соответствующий модуль приведён ниже:

```
UNIT Win_siz;

INTERFACE
Procedure Frame_w_ins(Wind,High,X_w,Y_w:integer);
Procedure Frame_w_del(Wind,High,X_w,Y_w:integer);

IMPLEMENTATION
uses CRT;
VAR
  Wind: integer; //  Ширина прямоугольника
  X_w: integer; //  X-координата прямоугольника
```

```
  Y_w: integer; //  Y-координата прямоугольника
 High: integer; //  Высота прямоугольника

{ Рисование прямоугольника:}
 Procedure Frame_w_ins;
  var i,j:byte;
 Begin
  GOTOXY(X_w-1,Y_w-1); Write(' ┌');
  FOR i:= 1 TO Wind do Write('─'); Write('┐ ');
  For j:= 0 to High Do
   Begin
    GOTOXY(X_w-1,Y_w+j); Write(' │ ');
    GOTOXY(X_w+Wind,Y_w+j); Write(' │ ');
   End;
  GOTOXY(X_w-1,Y_w+High+1); Write(' └');
  FOR i:= 1 TO Wind do Write('─'); Write('┘ ');
 End;

{ Стирание прямоугольника:}
 Procedure Frame_w_del;
  var i,j:byte;
 Begin
  GOTOXY(X_w-1,Y_w-1);
  FOR i:= 1 TO Wind+2 do Write(#32);
  For j:= 0 to High Do
   Begin
     GOTOXY(X_w-1,Y_w+j);
     FOR i:= 1 TO Wind+2 do Write(#32);
   End;
  GOTOXY(X_w-1,Y_w+High+1);
  FOR i:= 1 TO Wind+2 do Write(#32);
 End;

End.
```

Приведённый выше модуль **Win_siz** сохраняется в виде файла **Win_siz.pas**. Выполняя компиляцию данного модуля, мы получаем файл **Win_siz.tpu**, который сохраняется в папке **EXE**.

По завершении перечисленных выше действий, мы можем использовать процедуры и функции модуля **Win_siz**. Рассмотрим пример такого использования:

```
Program W_size;

uses CRT,WIN_SIZ; // Ссылка на модуль WIN_SIZ
VAR
  Wind: integer;  // Ширина прямоугольника
   X_w: integer;  // X-координата прямоугольника
   Y_w: integer;  // Y-координата прямоугольника
  High: integer;  // Высота прямоугольника

Begin
 CLRSCR;
 Wind:= 30; High:= 6; X_w:= 20; Y_w:= 3;
 Frame_w_ins(Wind,High,X_w,Y_w);
 Readln;
 Frame_w_del(Wind,High,X_w,Y_w);
 Readln;
End.
```

Запуская приведенную выше программу **W_size** на исполнение, получаем следующую картину:

Процедуры и функции модуля **Win_siz** можно использовать и более изощренным образом. В частности, можно организовать движение прямоугольника по экрану вдоль какой-либо траектории, например, вдоль овала. Рассмотрим соответствующую программу:

```
Program W_size;
uses CRT,WIN_SIZ; // Ссылка на модуль WIN_SIZ
VAR
  Wind: integer;  // Ширина прямоугольника
   X_w: integer;  // X-координата прямоугольника
   Y_w: integer;  // Y-координата прямоугольника
  High: integer;  // Высота прямоугольника

Begin
 CLRSCR;
 Wind:=8; High:=1; X_w:=2; Y_w:=0;
```

```
For k:= 1 to 240 Do
  begin
    X_w:= round(40+30*cos(k/12));
    Y_w:= round(12+10*sin(k/12));
    Frame_w_ins(Wind,High,X_w,Y_w);
    Delay(16000);
    Frame_w_del(Wind,High,X_w,Y_w);
  end;
End.
```

В приведенном примере мы наблюдаем движение прямоугольника по часовой стрелке вдоль овала. Результат этого движения можно сохранить, если не уничтожать следы. Добавим два небольших изменения:
Исключим процедуру, обеспечивающую стирание, заключив её в фигурные скобки:

**{Frame_w_del(Wind,High,X_w,Y_w);}**

и добавим оператор, меняющий цвет текста:

**TextColor(k mod 16)**

Результат выполнения программы приведён ниже:

**ПРИМЕР №2**. Моделирование работы стека.

Соответствующие модули приведёны ниже:

В модуле **LIFO_0** определена константа **max=10**, задающая максимальное количество элементов стека.

Тип **list=array[1..max] of string** задаёт массив строк, предназначенный для хранения элементов стека:

```
Unit LIFO_0;
Interface
const max=10;
type list=array[1..max] of string;
Implementation
end.
```

Процедура **LIFO_list** модуля **LIFO_1** предназначена для просмотра элементов стека. При просмотре элементы не меняются.

Следует отметить, что процедура **LIFO_list** модуля **LIFO_1** использует тип **list**. Это означает, что должен быть подключен модуль **LIFO_0**:

```
Unit LIFO_1;
```

```
Interface
uses CRT,LIFO_0;
var O: list; L: integer;
Procedure LIFO_list(O:list; L:integer);

Implementation
Procedure LIFO_list;
Var M: integer ;
  Ch: char ;
Begin
 GotoXY(25,20);
 Write('Elements view ...');
 GotoXY(25,22);
 If L>0 Then
   Begin M:=L;
    While M>0 do
      Begin
        Write(O[M],'  ');
        M:=M-1;
      End
   End Else Write('Empty, excuse ...') ;
 Ch:= ReadKey;
End;
end.
```

Процедура **LIFO_In** модуля **LIFO_2** предназначена для пополнения стека.

Следует отметить, что процедура **LIFO_In** модуля **LIFO_2** использует тип **list**. Это означает, что должен быть подключен модуль **LIFO_0**:

```
Unit LIFO_2;

Interface
uses CRT,LIFO_0;
var O: list; L: integer;
Procedure LIFO_In(var O:list; var L:integer);

Implementation
```

```
Procedure LIFO_In;
   Var Ch: char;
Begin
 GotoXY(25,20);
 Write('Record to Stack:');
 GotoXY(25,22);
 If L<max Then
  Begin
   L:=L+1;
   Str(Random(100),o[L]);
   Write(O[L])
  End
       Else Write('Full, excuse ...') ;
Ch:= ReadKey;
End;
End.
```

Процедура **LIFO_Out** модуля **LIFO_3** предназначена извлечения элементов из стека.

Следует отметить, что процедура **LIFO_Out** модуля **LIFO_3** использует тип **list**. Это означает, что должен быть подключен модуль **LIFO_0**:

```
Unit LIFO_3;

Interface
uses CRT,LIFO_0;
var O: list; L: integer;
Procedure LIFO_Out(var O:list; var L:integer);

Implementation
Procedure LIFO_Out;
  Var Ch: char;
Begin
  GotoXY(25,20);
  Write('Output elements:');
  GotoXY(25,22);
  If L>0 Then
   Begin
    Write('Output element: ', O[L]);
```

```
    L:=L-1
  End
      Else Write('Empty, excuse ... ') ;
 Ch:= ReadKey;
End;
End.
```

Процедура **MENU** модуля **LIFO_4** предназначена для вывода пунктов меню на экран и сохранения выбора пользователя (в переменной **Ch**).

```
Unit LIFO_4;

Interface
uses CRT;
Procedure MENU(var Ch:Char);

Implementation
Procedure MENU;
Begin
   GotoXY(30,7) ; Write('Stack modeling:') ;
   GotoXY(15,8); Write('(random meanings)') ;
   GotoXY(31,10); Write('1 - Stack listing');
   GotoXY(31,11); Write('2 - Stack record');
   GotoXY(31,12); Write('3 - Stack output');
   GotoXY(25,14); Write('Your choise: ');
   Ch:= ReadKey;
 End;
End.
```

В основной программе подключаются все приведенные выше модули. В цикле **Repeat ... until;** пользователю предоставляется возможность неограниченное число раз выполнять различные манипуляции со стеком.

Выход из программы - нажатие клавиши **ESC**.

```
Program LIFO_XXX;
uses CRT,
   LIFO_0,
   LIFO_1,
   LIFO_2,
   LIFO_3,
```

```
    LIFO_4;
var O: list;
  Ch: char;

Begin
 Randomize; CLRSCR;
 L:= 0 ;
 Repeat ClrScr; MENU(Ch);
 Case Ch of
  '1': LIFO_list(O,L);
  '2': LIFO_In(O,L);
  '3': LIFO_Out(O,L)
 End
 until Ch=#27
End.
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

**Вариант № 1.**

Задание №1.
Разработать программу, которая бы, используя модуль **WIN_SIZ.PAS**, обеспечивала перемещение прямоугольника из левого верхнего угла экрана в правый нижний.

Задание №2.
Используя
модули **LIFO_0.PAS**, **LIFO_1.PAS**, **LIFO_2.PAS**, **LIFO_3.PAS**, **LIFO_4.PAS**, смоделировать на базе массива стек, в который можно было бы заносить набираемые с клавиатуры числа.

Задание №3.
Составить, используя консольный режим работы Lazarus, проект, в котором бы находилась максимальная цифра вводимого с клавиатуры целого числа. Подпрограмма, обеспечивающая нахождение максимальной цифры, описывается в специально созданном для этого модуле.

**Вариант № 2.**

Задание №1.
Разработать программу, которая бы, используя модуль **WIN_SIZ.PAS**, обеспечивала перемещение прямоугольника из правого верхнего угла экрана в левый нижний.

Задание №2.
Используя
модули **LIFO_0.PAS**, **LIFO_1.PAS**, **LIFO_2.PAS**, **LIFO_3.PAS**, **LIFO_4.PAS**,

смоделировать на базе массива стек, в который можно было бы заносить случайные числа диапазона **50..150**.

Задание №3.
Составить, используя консольный режим работы Lazarus, проект, в котором бы находилась минимальная цифра вводимого с клавиатуры целого числа. Подпрограмма, обеспечивающая нахождение минимальной цифры, описывается в специально созданном для этого модуле.

**Вариант № 3.**

Задание №1.
Разработать программу, которая бы, используя модуль **WIN_SIZ.PAS**, обеспечивала перемещение прямоугольника из левого нижнего угла экрана в правый верхний.

Задание №2.
Используя
модули **LIFO_0.PAS**, **LIFO_1.PAS**, **LIFO_2.PAS**, **LIFO_3.PAS**, **LIFO_4.PAS**, смоделировать на базе массива стек, в который можно было бы заносить случайные числа диапазона 1**50..350** парами.

Задание №3.
Составить, используя консольный режим работы Lazarus, проект, в котором бы находилась сумма цифр вводимого с клавиатуры целого числа. Подпрограмма, обеспечивающая нахождение суммы цифр, описывается в специально созданном для этого модуле.

**Вариант № 4.**

Задание №1.
Разработать программу, которая бы, используя модуль **WIN_SIZ.PAS**, обеспечивала перемещение прямоугольника из правого нижнего угла экрана в левый верхний.

Задание №2.
Используя
модули **LIFO_0.PAS**, **LIFO_1.PAS**, **LIFO_2.PAS**, **LIFO_3.PAS**, **LIFO_4.PAS**, смоделировать на базе массива стек, в который можно было бы заносить случайные числа диапазона 20**0..2000** тройками.

Задание №3.
Составить, используя консольный режим работы Lazarus, проект, в котором бы находилось произведение цифр вводимого с клавиатуры целого числа. Подпрограмма, обеспечивающая нахождение произведения цифр, описывается в специально созданном для этого модуле.

**Вариант № 5.**

Задание №1.
Разработать программу, которая бы, используя модуль **WIN_SIZ.PAS**, обеспечивала перемещение прямоугольника слева направо так, чтобы его высота при этом увеличивалась.

Задание №2.
Используя
модули **LIFO_0.PAS**, **LIFO_1.PAS**, **LIFO_2.PAS**, **LIFO_3.PAS**, **LIFO_4.PAS**, смоделировать на базе массива стек, в который можно было бы заносить вводимые с клавиатуры пары чисел.

Задание №3.
Составить, используя консольный режим работы Lazarus, проект, в котором бы находилась сумма 3-х случайно сгенерированных чисел. Подпрограмма, обеспечивающая нахождение суммы этих чисел, описывается в специально созданном для этого модуле.

**Вариант № 6.**

Задание №1.
Разработать программу, которая бы, используя модуль **WIN_SIZ.PAS**, обеспечивала перемещение прямоугольника слева направо так, чтобы его высота при этом уменьшалась.

Задание №2.
Используя модули **LIFO_0.PAS**, **LIFO_1.PAS, LIFO_2.PAS, LIFO_3.PAS, LIFO_4.PAS**, смоделировать на базе массива стек, в который можно было бы заносить пары чисел. Одно из диапазона **40..340**, а второе - **100..300**.

Задание №3.
Составить, используя консольный режим работы Lazarus, проект, в котором бы находилось произведение 3-х случайно сгенерированных чисел. Подпрограмма, обеспечивающая нахождение произведения этих чисел, описывается в специально созданном для этого модуле.

**Вариант № 7.**

Задание №1.
Разработать программу, которая бы, используя модуль **WIN_SIZ.PAS**, обеспечивала перемещение прямоугольника сверху вниз.

Задание №2.
Используя модули **LIFO_0.PAS**, **LIFO_1.PAS, LIFO_2.PAS, LIFO_3.PAS, LIFO_4.PAS**, смоделировать на базе массива стек, в который можно было бы заносить пары чисел. Одно из диапазона 2**40..440**, а второе - **560..760**.

Задание №3.
Составить, используя консольный режим работы Lazarus, проект, в котором бы находилось максимальное из 3-х случайно сгенерированных чисел. Подпрограмма, обеспечивающая нахождение максимального из этих чисел, описывается в специально созданном для этого модуле.

**Вариант № 8.**

Задание №1.
Разработать программу, которая бы, используя модуль **WIN_SIZ.PAS**, обеспечивала перемещение прямоугольника снизу вверх.

Задание №2.

Используя модули **LIFO_0.PAS**, **LIFO_1.PAS, LIFO_2.PAS, LIFO_3.PAS, LIFO_4.PAS**, смоделировать на базе массива стек, в который можно было бы заносить меньшее из двух случайно сгенерированных чисел. Одно из диапазона 3**40..450**, а второе - **280..470**.

Задание №3.
Составить, используя консольный режим работы Lazarus, проект, в котором бы находилось произведение 2-х случайно сгенерированных чисел. Одно - из диапазона **200..300**, а второе - из диапазона **300..400**. Подпрограмма, обеспечивающая нахождение произведения этих чисел, описывается в специально созданном для этого модуле.

***ЛАБОРАТОРНОЕ ЗАНЯТИЕ № 3-4.***

*Тема:* Применение информационно-коммуникационных технологий с учетом основных требований информационной безопасности: Организация работы с объектами в среде программирования Паскаль

СТРУКТУРА ОБЪЕКТА

Ниже приводится описание структуры создаваемого объекта:

Кратчайшее описание структуры создаваемого объекта имеет следующий вид:

```
type
    Object_name = object
        <Поля и методы объекта>
    end;
```

В разделе **type** можно определить сразу несколько объектов. Например:

```
type
    Object_1 = object
        <Поля и методы объекта>
    end;
    Object_2 = object
        <Поля и методы объекта>
    end;
    Object_3 = object
        <Поля и методы объекта>
    end;
```

Поля и методы объектов определяются так:

```
type
    Object_name = object
    Private
        <Поля и методы объекта>
    Public
        <Поля и методы объекта>
    end;
```

Рассмотрим, как описанные выше определения объектов располагаются в модуле.

```
UNIT <Имя модуля>;
```

```
INTERFACE                    // Раздел объявления
                             // полей и методов объектов
type
 Object_name = object
 Private
. . . . . .
 Public
. . . . . .
 end;

IMPLEMENTATION      // Раздел описания объявленных
                    // в разделе INTERFACE
                    // методов объектов
<Описания методов объектов>
End.
```

## КОНСТРУИРОВАНИЕ ПРОСТЕЙШЕГО ОБЪЕКТА

Рассмотрим, как определяются и применяются объекты на конкретном примере:

Разработать объект, позволяющий решать следующую задачу: определять частичную сумму натурального ряда чисел, например:

**1 + 2 + 3 + … + n**

Данная задача является несложной, поэтому и объект, конструируемый для её решения, также должен быть несложным.

В конструируемый объект достаточно заложить один метод (суммирование членов ряда) и одно поле (сумма членов ряда).

Структура конструируемого нами объекта будет иметь следующий вид (красным цветом выделены вводимые нами имена:

**TSmrObj** - имя объекта;

**S**- имя единственного поля объекта;

**Summ** - имя единственного метода объекта, представляющего собою функцию, возвращающую значение искомой суммы;

**a** - имя параметра метода, обозначающего значение последнего слагаемого искомой суммы):

```
TSmrObj = object
Private
 S: integer;
```

```
Public
  Function Summ(a: integer):integer;
end;
```

В целом, модуль, содержащий описание конструируемого нами объекта, имеет следующий вид:

```
UNIT SMROW;

INTERFACE

type
 TSmrObj = object
 Private
  S: integer;
 Public
  Function Summ(a: integer):integer;
end;

IMPLEMENTATION

Function TSmrObj.Summ;
 var k: integer;
 begin
  S:= 0; k:= 1;
  while k <= a Do
   begin
    S:= S + k;
    k:= k + 1;
   end;
  Summ:= S;
 end;
End.
```

Данный модуль мы сохраняем в файле **SMROW.PAS**.

Следующий этап - компиляция созданного нами модуля, то есть файла **SMROW.PAS**.

В результате компиляции мы получаем файл **SMROW.TPU**.

Теперь создадим файл, в котором будут использоваться свойства объекта **TSmrObj**. При создании файла следует учитывать следующие три момента:

– модуль **SMROW.TPU** должен быть подключен командой **UsesSMROW**;

– объект **TSmrObj** представляет собою конструкцию, определённую в разделе **type**, поэтому мы обращаемся не к объекту **TSmrObj** непосредственно, а к переменной объектного типа, которая может быть определена, например, так:

```
sum: TSmrObj;
```

– обращение к методу **Summ** объекта **TSmrObj** осуществляется следующим образом:

**sum.Summ(n)**

В целом, программа, содержащая обращения к созданному объекту, может иметь следующий вид:

```
Program Summ_Row;
Uses SMROW;
var  n: integer;
  sum: TSmrObj;
Begin
 write('n= '); readln(n);
 writeln('Summa= ', sum.Summ(n));
 readln;
End.
```

Запуская приведённую выше программу на исполнение и вводя с клавиатуры значение переменной **n**, мы получаем на экране рассчитанное значение суммы:

**1 + 2 + 3 + … + n**

## КОНСТРУИРОВАНИЕ ПРОСТЕЙШЕГО ОБЪЕКТА-НАСЛЕДНИКА

Рассмотрим, как определяются и применяются объекты-наследники на конкретном примере:

Сконструировать объект-наследник, позволяющий решать следующую задачу: определять частичное произведение натурального ряда чисел, например:

**1 * 2 * 3 * … * n**

Для решения данной задачи сконструируем объект-наследник, который будет

– наследовать метод родительского объекта **TSmrObj**, обеспечивающий вычисление частичной суммы натурального ряда чисел;

– обладать методом вычисления частичного произведения натурального ряда чисел.

В конструируемый объект достаточно заложить один метод (нахождение произведения членов ряда) и одно поле (произведение членов ряда).

Структура конструируемого нами объекта будет иметь следующий вид (красным цветом выделены вводимые нами имена:

**TMtrObj** - имя объекта-наследника;

**TSmrObj** - имя объекта-родителя;
**P** - имя поля объекта, в котором хранится произведение;
**Mult** - имя метода объекта, представляющего собою функцию, возвращающую значение искомого произведения;
**a** - имя параметра метода, обозначающего значение последнего члена искомого произведения):

```
TMtrObj = object(TSmrObj)
 Public
  Function Mult(a: integer):integer;
 Private
  P: integer;
end;
```

В целом, модуль, содержащий описания конструируемыхнами объекта-потомка и его родителя, имеет следующий вид:

```
UNIT SUMUROW;

INTERFACE

type
 TSmrObj = object
 Public
  Function Summ(a: integer):integer;
 Private
  S: integer;
end;

 TMtrObj = object(TSmrObj)
 Public
  Function Mult(a: integer):integer;
Private
  P: integer;
end;

IMPLEMENTATION

Function TSmrObj.Summ;
 var k: integer;
 begin
  s:= 0; k:= 1;
  while k <= a Do
   begin
    S:= S + k;
    k:= k + 1;
   end;
  Summ:= S;
 end;

Function TMtrObj.Mult;
 var k: integer;
```

```
 begin
  p:= 1; k:= 1;
  while k <= a Do
   begin
    p:= p * k;
    k:= k + 1;
   end;
  Mult:= p;
 end;

End.
```

Данный модуль мы сохраняем в файле **SUMUROW.PAS**.

Следующий этап - компиляция созданного нами модуля, то есть файла **SUMUROW.PAS**.

В результате компиляции мы получаем файл **SUMUROW.TPU**.

Завершив компиляцию модуля, приступим к созданию файла, в котором будут использоваться методы как объекта-родителя **TSmrObj**, так и объекта-потомка **TMtrObj**. При создании файла следует учитывать следующие моменты:

– переменные объектного типа определим следующим образом:

```
sum: TSmrObj;
sumu: TMtrObj;
```

– обращение к методу **Summ** объекта **TSmrObj** осуществляется следующим образом:

```
sum.Summ(n);
```

– обращение к унаследованному методу **Summ** и собственному методу **Mult** объекта **TMtrObj** осуществляется следующим образом:

```
sumu.Summ(n);
sumu.Mult(n);
```

В целом, программа, содержащая обращения к созданному объекту, может иметь следующий вид:

```
Program Sumu_Row;
Uses SUMUROW;
var  n: integer;
   sum: TSmrObj;
  sumu: TMtrObj;
Begin
 write('n= '); readln(n);
 writeln('Smr.summa= ',sum.Summ(n));
 writeln('Mtr.summa= ',sumu.Summ(n));
 writeln('Mtr.mult= ',sumu.Mult(n));
 readln;
```

```
End.
```

Запуская приведённую выше программу на исполнение и вводя с клавиатуры значение переменной **n**, мы получаем на экране значения (**1+2+3+…+n**) и (**1*2*3*…*n**):

```
n= 5
Smr.summa= 15
Mtr.summa= 15
Mtr.mult= 120
```

Метод **Summ** объектом-наследником (**TMtrObj**) унаследован от объекта-родителя (**TSmrObj**), поэтому полученные значения сумм, естественно, совпадают.

## КОНСТРУИРОВАНИЕ ПРОСТЕЙШЕГО ОБЪЕКТА-НАСЛЕДНИКА, ПЕРЕОПРЕДЕЛЯЮЩЕГО МЕТОД ОБЪЕКТА-РОДИТЕЛЯ

Рассмотрим, как осуществляется переопределение метода объекта-родителя объектом-наследником на конкретном примере:

Сконструировать объект-наследник, позволяющий решать следующие задачи:

определять частичное произведение натурального ряда чисел, например:

**1 * 2 * 3 * … * n**

и определять частичную сумму чисел натурального ряда, но не всех чисел, а только четных:

**2 + 4 + 6 + …**

Для решения данной задачи сконструируем объект-наследник, который будет:

– переопределять метод родительского объекта **TSmrObj**, обеспечивающий вычисление частичной суммы натурального ряда чисел;

– обладать методом вычисления частичного произведения натурального ряда чисел.

В конструируемый объект достаточно заложить два метода (нахождение произведения членов ряда и нахождение суммы четных чисел ряда).

Структура конструируемого нами объекта будет иметь следующий вид (красным цветом выделены вводимые нами имена:

**TMtrObj** - имя объекта-наследника;

**TSmrObj** - имя объекта-родителя;

**Mult** - имя метода объекта, представляющего собою функцию, возвращающую значение искомого произведения;

**Summ** - имя метода объекта, представляющего собою функцию, возвращающую значение суммы четных членов ряда;

**a** - имя параметра метода, обозначающего значение последнего члена искомого произведения):

```
TMtrObj = object(TSmrObj)
 Public
  Function Mult(a: integer):integer;
  Function Summ(a: integer):integer;
 Private
end;
```

Из определения следует, что объект-наследник не имеет ни одного поля. Последнее означает, что рассчитанные значения объект не сохраняет, а сразу же возвращает (например, выводит на экран монитора).

В целом, модуль, содержащий описания конструируемыхнами объекта-потомка и его родителя, имеет следующий вид:

```
UNIT SUMUROW;

INTERFACE

type
 TSmrObj = object
 Public
  Function Summ(a: integer):integer;
 Private
  S: integer;
end;

 TMtrObj = object(TSmrObj)
 Public
  Function Mult(a: integer):integer;
  Function Summ(a: integer):integer;
 Private
end;

IMPLEMENTATION

Function TSmrObj.Summ;
 var k: integer;
 begin
  s:= 0; k:= 1;
  while k <= a Do
   begin
    S:= S + k;
    k:= k + 1;
   end;
  Summ:= S;
```

```
 end;

Function TMtrObj.Mult;
 var k, p: integer;
 begin
  p:= 1; k:= 1;
  while k <= a Do
   begin
    p:= p * k;
    k:= k + 1;
   end;
  Mult:= p;
 end;

Function TMtrObj.Summ;
 var k, s: integer;
 begin
  s:= 0; k:= 1;
  while k <= a Do
   begin
    if k mod 2 = 0 then s:= s + k;
    k:= k + 1;
   end;
  Summ:= s;
 end;

End.
```

Данный модуль мы сохраняем в файле **SUMUROW.PAS**.

В результате компиляции файла **SUMUROW.PAS** мы получаем файл **SUMUROW.TPU**.

Создание файла, в котором будут использоваться методы объекта-родителя **TSmrObj** и объекта-потомка **TMtrObj** осуществляется так, как это было описано выше.

В целом, программа, содержащая обращения к методам объектов (и родителя, и потомков), может иметь следующий вид:

```
Program Sumu_Row;
Uses SUMUROW;
var  n: integer;
   sum: TSmrObj;
  sumu: TMtrObj;
Begin
 write('n= '); readln(n);
 writeln('Smr.summa= ',sum.Summ(n));
 writeln('Mtr.summa= ',sumu.Summ(n));
 writeln('Mtr.mult= ',sumu.Mult(n));
 readln;
End.
```

Запуская приведённую выше программу на исполнение и вводя с клавиатуры значение переменной **n**, мы получаем на экране:

```
n= 5
Smr.summa= 15
Mtr.summa= 6
Mtr.mult= 120
```

Метод **Summ** объекта-родителя (**TSmrObj**) в объекте-наследнике (**TMtrObj**) переопределён, поэтому полученные значения сумм, естественно, различаются.

Рассмотрим более сложные примеры реализации объектов.

ПРИМЕР №1. Прорисовка прямоугольника на алфавитно-цифровом экране.
ПРИМЕР №2. Моделирование работы стека.

**ПРИМЕР №1**. Прорисовка прямоугольника на алфавитно-цифровом экране.

Выполнять в консольном приложении Lazarus (Delphi).

Соответствующий модуль приведён ниже:

```
unit Obj_siz;

interface
uses Windows;
type
TWinSiz = object
Public
 Procedure Frame_w_ins(Wind,High,X_w,Y_w:integer);
 Procedure Frame_w_del(Wind,High,X_w,Y_w:integer);

Private
end;
procedure clrscr;

implementation
procedure clrscr;
var
```

```
  cursor: COORD;
  r: cardinal;
begin
  r := 300;
  cursor.X := 0;
  cursor.Y := 0;
  FillConsoleOutputCharacter(GetStdHandle(STD_OUTPUT_HANDLE), ' ', 80 * r,
cursor, r);
  SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), cursor);
end;

Procedure TWinSiz.Frame_w_ins;
 var i,j:byte;
Begin

 for i:=1 to Y_w  do writeln;
 for i:=1 to X_w  do write(' ');
 FOR i:= 1 TO Wind+1 do Write(#219);
 writeln;
 For j:= 1 to High Do
 Begin
 for i:=1 to X_w  do write(' ');Write(#219);
 FOR i:= 1 TO Wind-1 do Write(' '); Write(#219);
 writeln;
 End;
 for i:=1 to X_w  do write(' ');
 FOR i:= 1 TO Wind+1 do Write(#219);
End;
{ Erase Rectangle: }
Procedure TWinSiz.Frame_w_del;
var i,j:byte;
Begin
 clrscr;
End;



end.
```

Прорисовка прямоугольника на алфавитно-цифровом экране.

Выполнять в консольном приложении PascalABC.

Соответствующий модуль приведён ниже:

```
UNIT Obj_siz;
INTERFACE
type  TWinSiz = class
Public
Procedure Frame_w_ins(Wind,High,X_w,Y_w:integer);
Procedure Frame_w_del(Wind,High,X_w,Y_w:integer);
Private end;
IMPLEMENTATION
uses CRT;
{ Draw Rectangle:}
Procedure TWinSiz.Frame_w_ins(Wind,High,X_w,Y_w:integer);
var i,j:byte;
Begin
 GOTOXY(X_w - 1,Y_w - 1);
 Write('┌');
 FOR i:= 1 TO Wind do Write('─'); Write('┐');
 For j:= 0 to High Do
 Begin
 GOTOXY(X_w - 1,Y_w+j); Write('│');
 GOTOXY(X_w + Wind,Y_w+j); Write('│');
 End;   GOTOXY(X_w - 1,Y_w + High + 1); Write('└');
 FOR i:= 1 TO Wind do Write('─'); Write('┘');
 End;
 { Erase Rectangle:}
 Procedure TWinSiz.Frame_w_del(Wind,High,X_w,Y_w:integer);
```

```
var i,j:byte;
Begin   GOTOXY(X_w - 1,Y_w - 1);
FOR i:= 1 TO Wind+2 do Write(#32);
For j:= 0 to High Do
Begin
GOTOXY(X_w - 1,Y_w+j);
FOR i:= 1 TO Wind+2 do Write(#32);
End;   GOTOXY(X_w - 1,Y_w + High + 1);
FOR i:= 1 TO Wind+2 do Write(#32);
End;
End.
```

Приведённый выше модуль **Obj_siz** сохраняется в виде файла **Obj_siz.pas**.
Выполняя компиляцию данного модуля, мы получаем файл **Obj_siz.tpu**.
По завершении перечисленных выше действий, мы можем использовать методы объекта **TWinSiz**, определённого в модуле **Obj_siz**. Рассмотрим пример такого использования:

```
Program Obj_siz1;

uses  Obj_siz;

VAR
  Ows: TWinSiz;
 Wind: integer; { Rectangle's width}
  X_w: integer; { Rectangle's X-coordinates}
  Y_w: integer; { Rectangle's Y-coordinates}
 High: integer; { Rectangle's width}
   k: integer;
Begin
Ows:=TWinSiz.create;
CLRSCR;
 Wind:= 10; High:= 2; X_w:= 10; Y_w:= 10;
```

```
For k:= 1 to 10 Do
  begin X_w:= X_w+4;
   Ows.Frame_w_ins(Wind, High, X_w, Y_w);
   Readln;
   Ows.Frame_w_del(Wind,High,X_w,Y_w);
  end;
Readln;
End.
```

Запуская приведенную выше программу **Obj_siz1** на исполнение, мы получаем прямоугольник, который можно перемещать по горизонтали слева направо нажатием клавиши **Enter** (10 нажатий):

**ПРИМЕР №2**. Моделирование работы стека.

Соответствующий модуль приведён ниже:

В модуле **Obj_LIFO** определена константа **max=10**, задающая максимальное количество элементов стека.

Тип **list=array[1..max] of string** задаёт массив строк, предназначенный для хранения элементов стека.

Объект **TLIFO** имеет следующую структуру:

- два поля (**O** - массив элементов стека и **L** - количество элементов стека):
- пять методов (**Init_list**, **LIFO_list**, **LIFO_In**, **LIFO_Out**, **MENU**), описания которых приводятся ниже.

Процедура **Init_list** поддерживает метод, предназначенный для начальной инициализации стека (количество элементов стека устанавливается = 0).

Процедура **LIFO_list** поддерживает метод, предназначенный для просмотра элементов стека. При просмотре элементы не меняются.

Процедура **LIFO_In** поддерживает метод, предназначенный для пополнения стека.

Процедура **LIFO_Out** поддерживает метод, предназначенный для извлечения элементов из стека.

Процедура **MENU** поддерживает метод, предназначенный для вывода пунктов меню на экран и сохранения выбора пользователя.

```
Unit Obj_LIFO;

Interface

const max = 10;
type list = array[1..max] of string;
```

```
TLIFO = object
 Public
  Constructor create;
  Procedure LIFO_list;
  Procedure LIFO_In;
  Procedure LIFO_Out;
  Procedure MENU(var Ch:Char);
 Private
  O: list;
  L: integer;
end;

Implementation

uses CRT;

{ Initialization TLIFO:}
Constructor TLIFO.create;
Begin
  inherited create;
  L:= 0;
 end;

Procedure TLIFO.LIFO_list;
Var M: integer;
   Ch: char;
Begin
  GotoXY(25,20); Write('Elements view ...');
  GotoXY(25,22) ;
  If L>0 Then
     Begin M:=L;
      While M>0 do
        Begin Write(O[M],'  '); M:=M-1; End
    End Else Write('Empty, excuse ...') ;
  Ch:= ReadKey;
```

```
End;

Procedure TLIFO.LIFO_In;
   Var Ch: char;
Begin
 GotoXY(25,20); Write('Record to Stack:');
 GotoXY(25,22);
 If L<max Then
   Begin
    L:=L+1; Str(Random(100),o[L]); Write(O[L])
   End
        Else Write('Full, excuse ...') ;
Ch:= ReadKey;
End;

Procedure TLIFO.LIFO_Out;
  Var Ch: char;
Begin
  GotoXY(25,20); Write('Output elements:');
  GotoXY(25,22);
  If L>0 Then
   Begin
    Write('Output element: ', O[L]); L:=L-1
   End
       Else Write('Empty, excuse ... ') ;
  Ch:= ReadKey;
End;

Procedure TLIFO.MENU;
Begin
   GotoXY(30,7) ; Write('Stack modeling:');
   GotoXY(15,8); Write('(random meanings)');
   GotoXY(31,10); Write('1 - Stack listing');
   GotoXY(31,11); Write('2 - Stack record');
   GotoXY(31,12); Write('3 - Stack output');
   GotoXY(25,14); Write('Your choise: ');
```

```
   Ch:= ReadKey;
 End;


end.
```

В основной программе подключается модуль **Obj_LIFO**, что даёт возможность использовать определённые в этом модуле методы объекта **TLIFO**.

В цикле **Repeat ... until;** пользователю предоставляется возможность неограниченное число раз выполнять различные манипуляции со стеком.

Выход из программы - нажатие клавиши **ESC**.

```
Program LIFO_OBJ;
uses CRT, OBJ_LIFO;

var  Ch: char;
   LIFO: TLIFO;
Begin
  LIFO:=TLIFO.create;
  Randomize; CLRSCR;
  Repeat
    ClrScr;
    LIFO.MENU(Ch);
    Case Ch of
     '1': LIFO.LIFO_list;
     '2': LIFO.LIFO_In;
     '3': LIFO.LIFO_Out
    End
  until Ch=#27
End.
```

## ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

**Вариант № 1.**

Задание №1.
Разработать программу, которая бы, используя объект **TWinSiz**, определенный в модуле **Obj_siz.PAS**, обеспечивала перемещение прямоугольника из левого верхнего угла экрана вправый нижний.

Задание №2.
Используя объект **TLIFO.PAS**, определенный в модуле **Obj_LIFO.PAS**,

смоделировать стек, в который можно было бы заносить набираемые с клавиатуры числа.

**Вариант № 2.**

Задание №1.
Разработать программу, которая бы, используя объект **TWinSiz**, определенный в модуле **Obj_siz.PAS**, обеспечивала перемещение прямоугольника из правого верхнего угла экрана влевый нижний.

Задание №2.
Используя объект **TLIFO.PAS**, определенный в модуле **Obj_LIFO.PAS**, смоделировать стек, в который можно было бы заносить случайные числа диапазона **50..150**.

**Вариант № 3.**

Задание №1.
Разработать программу, которая бы, используя объект **TWinSiz**, определенный в модуле **Obj_siz.PAS**, обеспечивала перемещение прямоугольника из левого нижнего угла экрана вправый верхний.

Задание №2.
Используя объект **TLIFO.PAS**, определенный в модуле **Obj_LIFO.PAS**, смоделировать стек, в который можно было бы заносить случайные числа диапазона 1**50..350** парами.

**Вариант № 4.**

Задание №1.
Разработать программу, которая бы, используя объект **TWinSiz**, определенный в модуле **Obj_siz.PAS**, обеспечивала перемещение прямоугольника из правого нижнего угла экрана влевый верхний.

Задание №2.
Используя объект **TLIFO.PAS**, определенный в модуле **Obj_LIFO.PAS**, смоделировать стек, в который можно было бы заносить случайные числа диапазона 20**0..2000** тройками.

**Вариант № 5.**

Задание №1.
Разработать программу, которая бы, используя объект **TWinSiz**, определенный в модуле **Obj_siz.PAS**, обеспечивала перемещение прямоугольника слева направо так, чтобы его высота при этом увеличивалась.

Задание №2.
Используя объект **TLIFO.PAS**, определенный в модуле **Obj_LIFO.PAS**, смоделировать стек, в который можно было бы заносить вводимые с клавиатуры пары чисел.

**Вариант № 6.**

Задание №1.
Разработать программу, которая бы, используя объект **TWinSiz**,

определенный в модуле **Obj_siz.PAS**, обеспечивала перемещение прямоугольника слева направо так, чтобы его высота при этом уменьшалась.

Задание №2.
Используя объект **TLIFO.PAS**, определенный в модуле **Obj_LIFO.PAS**, смоделировать стек, в который можно было бы заносить пары чисел. Одно из диапазона **40..340**, а второе - **100..300**.

**Вариант № 7.**

Задание №1.
Разработать программу, которая бы, используя объект **TWinSiz**, определенный в модуле **Obj_siz.PAS**, обеспечивала перемещение прямоугольника сверху вниз.

Задание №2.
Используя объект **TLIFO.PAS**, определенный в модуле **Obj_LIFO.PAS**, смоделировать стек, в который можно было бы заносить пары чисел. Одно из диапазона 2**40..440**, а второе - **560..760**.

**Вариант № 8.**

Задание №1.
Разработать программу, которая бы, используя объект **TWinSiz**, определенный в модуле **Obj_siz.PAS**, обеспечивала перемещение прямоугольника снизу вверх.

Задание №2.
Используя объект **TLIFO.PAS**, определенный в модуле **Obj_LIFO.PAS**, смоделировать стек, в который можно было бы заносить меньшее из двух случайно сгенерированных чисел. Одно из диапазона 3**40..450**, а второе - **280..470**.

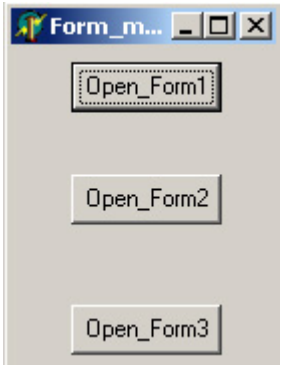
***ЛАБОРАТОРНОЕ ЗАНЯТИЕ № 5-7.***

*Тема:* Разработка моделей интерфейсов «человек - электронно-вычислительная машина»: Технология программирования в оконных операционных средах

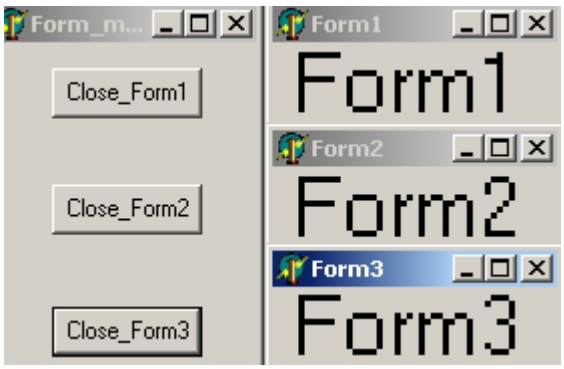
## СОЗДАНИЕ LAZARUS-ПРОЕКТА, ВКЛЮЧАЮЩЕГО В СВОЙ СОСТАВ НЕСКОЛЬКИХ ФОРМ.

Предположим, следует создать проект, позволяющий связать с событием **OnClick**появление/исчезновение дополнительных форм.

После запуска приложения на экране должна отобразиться исходная форма:




Щелчки по кнопкам должны привести к отображению на экране всех форм проекта:




Приступим к созданию проекта, позволяющего выполнять указанные выше действия. Последовательность этих действий распишем по шагам.

Шаг 1.

Создаём новый проект и сохраняем его в какой-либо специально для этого созданной папке. При сохранении проекта задаём его имя, например, **Project_1** и имя модуля, например, **Unit_main**. В результате проект будет сохранён в файле **Project_1.dpr**, а модуль - в файле **Unit_main.pas**.

На данном этапе автоматически сгенерированный файл проекта **Project_1.dpr** будет иметь следующий вид:

```
program Project_1;
uses
  Forms,
  Unit_main in 'Unit_main.pas' {Form_main},
{$R *.RES}
begin
```

```
  Application.Initialize;
  Application.CreateForm(TForm_main, Form_main);
  Application.Run;
end.
```

В свою очередь, файл **Unit_main.pas** будет иметь вид:

```
unit Unit_main;

interface

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

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

var
  Form_main: TForm_main;

implementation

{$R *.DFM}

end.
```

Конструкции **TForm_main** и **Form_main** генерируются автоматически, без участия программиста.

Шаг 2.

На пустой форме проекта создаём 3 кнопки: **Button1**, **Button2** и **Button3**.

В результате раздел **type** модуля, открывающийся строкой

**TForm_main = class(TForm)**,

принимает следующий вид:

```
  TForm_main = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Шаг 3.

Создаём обработчик события **OnCreate** (то есть создаём процедуру, которая запускается, когда наступает событие **OnCreate**), связанного с созданием основной

формы.

В результате раздел **type** модуля принимает вид:

```
 TForm_main = class(TForm)
  Button1: TButton;
  Button2: TButton;
  Button3: TButton;
  procedure FormCreate(Sender: TObject);
 private
  { Private declarations }
 public
  { Public declarations }
 end;
```

Описание процедуры **FormCreate** генерируется автоматически и размещается в разделе **implementation** модуля:

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

end;
```

Пока процедура пуста, то есть никакой обработки события не происходит.

Шаг 4.

Введём в процедуру **FormCreate** код. Этот код не генерируется автоматически. Его необходимо вводить в ручном режиме.

```
Width:=140;       // задание ширины формы
Height:=300;      // задание высоты формы
Top:=100;         // задание верхнего отступа формы от края экрана
Left:=300;        // задание левого отступа формы от края экрана
Button1.Top:=30;  // задание верхнего отступа 1-ой кнопки от края формы
Button1.Left:=30; // задание левого отступа 1-ой кнопки от края формы
Button2.Top:=130; // задание верхнего отступа 2-ой кнопки от края формы
Button2.Left:=30; // задание левого отступа 2-ой кнопки от края формы
Button3.Top:=230; // задание верхнего отступа 3-ей кнопки от края формы
Button3.Left:=30; // задание левого отступа 3-ей кнопки от края формы
Button1.Tag:=0;   // задание значения свойства Tag 1-ой кнопки
Button2.Tag:=0;   // задание значения свойства Tag 2-ой кнопки
Button3.Tag:=0;   // задание значения свойства Tag 3-ей кнопки
```

Свойство **Tag** кнопок потребуется для управления видимостью вторичных форм, которые будут созданы в дальнейшем.

Шаг 5.

Теперь включим в состав проекта три дополнительные формы. В результате файл проекта **Project_1.dpr**, принимает вид:

```
program Project_1;
uses
  Forms,
  Unit_main in 'Unit_main.pas' {Form_main},
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2},
  Unit3 in 'Unit3.pas' {Form3};
{$R *.RES}
```

```
begin
  Application.Initialize;
  Application.CreateForm(TForm_main, Form_main);
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.CreateForm(TForm3, Form3);
  Application.Run;
end.
```

Файл модуля, например, **Unit_1**, сохраняется в файле **Unit_1.pas**:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation
{$R *.DFM}

end.
```

Для использования конструкций (пока эти конструкции не созданы) трёх подключенных к проекту модулей (**Unit_1**, **Unit_2**, **Unit_3**) в раздел **uses** файла **Unit_main.pas** следует внести коррективы:

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls,Unit1, Unit2, Unit3;
```

Теперь из основного модуля можно будет управлять конструкциями, которые будут созданы в модулях **Unit1**, **Unit2**, **Unit3**.

Шаг 6.

Создаём обработчик события **OnCreate**, связанного с созданием вторичных форм, например, формы **Form1**:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Width:= 150;  // задание ширины формы
 Height:= 100; // задание высоты формы
 Top:= 100;    // задание верхнего отступа формы от края экрана
 Left:= 440;   // задание левого отступа формы от края экрана
end;
```

Обращаем внимание на согласованность параметров первичной формы **Form_main** и вторичных форм (**Form1**, **Form2**, **Form3**):

**1-ое согласование:**

**Form_main.Top = Form1.Top = 100**

**2-ое согласование (для каждой из 3-х вторичных форм):**

**Form1.Left= Form_main.Left + Form_main.Width = 300+140= 440**

**3-е согласование:**

**Form_main.Height= Form1.Height + Form2.Height + Form3.Height=**

**100 + 100 + 100 = 300**

**4-ое согласование:**

**Form2.Top= Form1.Top + Form1.Height= 100 + 100= 200**

**5-ое согласование:**

**Form3.Top= Form2.Top + Form2.Height= 200 + 100= 300**

Шаг 7.

Создаём обработчик события **OnPaint**, связанного с выведением текста на вторичные формы, например, форму **Form1**:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Brush.Style:= bsClear;  // очистка формы
 Canvas.Font.Size:= 30;         // задание размера шрифта
 Canvas.TextOut(10,10,'Form1')  // задание текста и его расположения
end;
```

После создания обработчиков событий **OnCreate** и **OnPaint** модуль, например, **Unit_1**, сохраняемый в файле **Unit_1.pas**, выглядит так:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Width:=150;
 Height:=100;
 Top:=100;
 Left:=440;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Brush.Style:=bsClear;
 Canvas.Font.Size:=30;
 Canvas.TextOut(10,10,'Form1')
end;

end.
```

Шаг 8.

Создаём обработчики события **OnClick**, связанного со щелчками по каждой из 3-х кнопок основной формы **Form_main**.

В результате раздел **type** модуля **Unit_main** принимает вид:

```
  TForm_main = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Описание процедур (например, процедуры **Button1Click**), генерируются автоматически и размещаются в разделе **implementation** модуля, например:

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

end;
```

На данном этапе все 3 созданных нами процедуры пусты.

Шаг 9.

Введём в процедуры **Button1Click**, **Button2Click**, **Button3Click** код. Код вводится вр учном режиме. Рассмотрим код процедуры **Button1Click**:

```
procedure TForm_main.Button1Click(Sender: TObject);
begin
 with Button1 Do  // задание свойств кнопки Button1
  If Tag=0 Then   // задание свойств кнопки, если Tag=0:
    Begin
      Form1.Show; // визуализация формы Form1
```

```
      Tag:=1; // изменение значения Tag, т.к. форма Form1 стала видимой
      Caption:= 'Close_Form1' // изменение заголовка кнопки
    End
        Else   // задание свойств кнопки, если Tag=1:
    Begin
      Form1.Hide; // сокрытие формы Form1
      Tag:=0; // изменение значения Tag, т.к. форма Form1 стала невидимой
      Caption:= 'Open_Form1' // изменение заголовка кнопки
    End
  end;
```

Шаг 10.

После создания обработчиков событий **OnClick** модуль **Unit_main**, выглядит так:

```
unit Unit_main;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Unit1, Unit2, Unit3;

type
  TForm_main = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form_main: TForm_main;

implementation
{$R *.DFM}

procedure TForm_main.Button1Click(Sender: TObject);
begin
 with Button1 Do
  If Tag=0 Then
    Begin
      Form1.Show; Tag:=1; Caption:= 'Close_Form1'
    End
        Else
    Begin
      Form1.Hide; Tag:=0; Caption:= 'Open_Form1'
```

```
   End
end;

procedure TForm_main.Button2Click(Sender: TObject);
begin
 with Button2 Do
  If Tag=0 Then
   Begin
    Form2.Show; Tag:=1; Caption:= 'Close_Form2'
   End
       Else
   Begin
    Form2.Hide; Tag:=0; Caption:= 'Open_Form2'
   End
end;

procedure TForm_main.Button3Click(Sender: TObject);
begin
 with Button3 Do
  If Tag=0 Then
   Begin
    Form3.Show; Tag:=1; Caption:= 'Close_Form3'
   End
       Else
   Begin
    Form3.Hide; Tag:=0; Caption:= 'Open_Form3'
   End
end;

procedure TForm_main.FormCreate(Sender: TObject);
begin
 Width:=140; Height:=300;
 Top:=100;   Left:=300;
 Button1.Top:=30;  Button1.Left:=30; Button1.Tag:=0;
 Button2.Top:=130; Button2.Left:=30; Button2.Tag:=0;
 Button3.Top:=230; Button3.Left:=30; Button3.Tag:=0;
end;

end.
```

Результаты работы сохраняем в файлах проекта (**Project_1.dpr**, **Unit_main.pas**, **Unit_1.pas**, **Unit_2.pas**, **Unit_3.pas**).

## ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

**Вариант № 1.**

Задание.

Взяв за основу **Project_1**, разработать проект, в котором каждая из визуализируемых форм (**Form1**, **Form2**, **Form3**) имела бы собственный цвет.

**Вариант № 2.**

Задание.
Взяв за основу **Project_1**, разработать проект, в котором каждая из визуализируемых форм (**Form1**, **Form2**, **Form3**) в стартовом положении была бы видимой.

**Вариант № 3.**

Задание.
Взяв за основу **Project_1**, разработать проект, в котором каждая из визуализируемых форм (**Form1**, **Form2**, **Form3**) не имела бы текстовых надписей **Form1**, **Form2**, **Form3**.

**Вариант № 4.**

Задание.
Взяв за основу **Project_1**, разработать проект, в котором каждая из визуализируемых форм (**Form1**, **Form2**, **Form3**) имела бы текстовую надпись (**Form1**, **Form2**, **Form3**) оригинального цвета.

**Вариант № 5.**

Задание.
Взяв за основу **Project_1**, разработать проект, в котором цвета надписей на кнопках (**Close_Form** и **Open_Form**) были бы различными.
Указание: кнопки типа **Button** заменить кнопками типа **BitBtn**.

**Вариант № 6.**

Задание.
Взяв за основу **Project_1**, разработать проект, в котором основная форма **Form_main** меняла бы собственный цвет при щелчках по кнопкам.

**Вариант № 7.**

Задание.
Взяв за основу **Project_1**, разработать проект, в котором надписи на кнопках **Close_Form** и **Open_Form** менялись бы свой цвет при щелчках по ним.
Указание: кнопки типа **Button** заменить кнопками типа **BitBtn**.

**Вариант № 8.**

Задание.
Взяв за основу **Project_1**, разработать проект, в котором каждая из визуализируемых форм (**Form1**, **Form2**, **Form3**) меняла бы собственный цвет при щелчках по соответствующим кнопкам.

## Функции. Пример использования.

Рассмотрение начнём с примера простейшей функции (**summ**), возвращающей сумму аргументов **(a,b)**, подаваемых на её вход (**summ:= a+b**). Возвращаемое функцией значение может быть выведено на экран или присвоено переменной соответствующего типа.

Листинг 1.1.

```
Unit Summa;
interface
  Function summ(a,b: real): real;
implementation
  Function summ;
   begin
     summ:= a+b;
   end;
end.
```

Функция (**summ**) определена в модуле **Summa** (текст модуля, фигурирующий в листинге 1.1, хранится в файле с тем же именем - "**Summa.pas**"). Скелет структуры модуля **Summa** приведен ниже (листинг 1.2).

Листинг 1.2.

```
Unit Summa;
interface
implementation
end.
```

В разделе **interface** модуля определяются заголовки функций и процедур (в данном случае - единственной функции **summ**). Развёрнутые определения функций и процедур приводятся в разделе **implementation** модуля.

Следует обратить внимание на то, что и в **Turbo Pascal**, и в консольных приложениях **Lazarus** заголовки определений функций приводятся полностью ("**Function summ(a,b: real):real**"), в расшифровках же заголовки приводятся в сокращенной нотации ("**Function summ**").

Использование определённой в модуле **Summa** функции **summ** в **Turbo Pascal**, и в консольных приложениях **Lazarus** организовано почти идентично.

В **Turbo Pascal** использование функции **summ** может быть реализовано следующим образом:

Листинг 1.3.

```
Program Summa_xy;
uses Summa;
var x,y: real;
Begin
 Write('x= '); Readln(x);
 Write('y= '); Readln(y);
 Writeln('x+y= ',Summ(x,y):4:2);
 Readln;
End.
```

Краткий комментарий к приведённому выше коду:

- различие в заголовках ("**Unit Summa;**" и "**Program Summa_xy;**");
- обязательное подключение ресурсов модуля **Summa** ("**uses Summa;**");
- обязательное определение аргументов функции **Summ** ("**var x,y:real;**");
- введение значений аргументов с клавиатуры ("**Readln(x)**" и "**Readln(y)**");
- заключительное обращение к функции **Summ** ("**Summ(x,y)**").

В консольном приложении **Lazarus** единственное отличие от приведённого выше кода - подсказка компилятору **Lazarus**, что перед ним - приложение консольного типа:

**{$APPTYPE CONSOLE}**

Эта строка ставится сразу после заголовка программы("**Program Summa_xy;**").

## ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Определить в модуле **Summa** ещё одну функцию (**Diff(a,b)**), которая бы возвращала разность поданных на её вход аргументов. В программу (**Program Summa_xy**) внести изменения, позволяющие использовать функцию **Diff**.

2. Создать новый модуль **Multipl**, в котором определить функцию (**Mult(a,b)**), возвращающую произведение поданных на её вход аргументов. Создать программу (**Program Mult_xy**), позволяющую использовать функцию **Mult**.

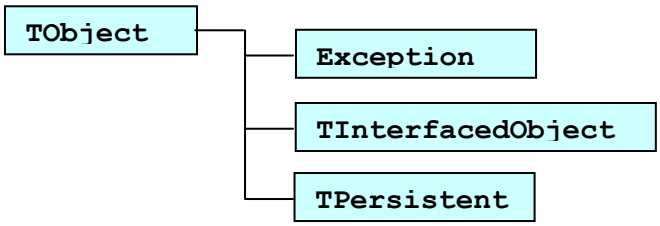
3. Создать новый модуль **Big_Summ**, в котором определить функцию (**Summ(a,b,c)**), возвращающую сумму поданных на её вход аргументов. Создать программу (**Program Summ_xyz**), позволяющую использовать функцию **Summ**.

Примечание. Задания можно выполнять как в среде программирования **Turbo Pascal** (**Borland Pascal**), так и в среде программирования **Lazarus** (создание консольного приложения).

***ЛАБОРАТОРНОЕ ЗАНЯТИЕ № 8-10.***

*Тема:* Определение простейшего класса в Lazarus

Рассмотрение начнём с приведения фрагмента дерева классов в **Lazarus**.




Основополагающим, корневым, базовым классом в иерархии классов **Lazarus** является класс **TObject**.

Класс **TObject** определяется в модуле **System**. Данный модуль не упоминается в перечне модулей, фигурирующих после ключевого слова **uses** (полная аналогия с **Turbo** и **Borland Pascal**). Ресурсы этого модуля на стадии компиляции подключаются автоматически.

Методы класса **TObject** могут использоваться любыми классами-потомками. Перечислим в качестве примера ряд методов класса **TObject**:

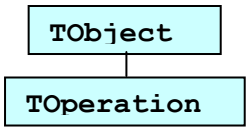
- метод **Create** - возвращает экземпляр (объект) соответствующего класса;
- метод **ClassType** - возвращает ссылку на класс объекта;
- метод **ClassName** - возвращает имя класса;
- метод **ClassParent** - возвращает ссылку на родительский класс;
- метод **InstanceSize** - возвращает размер объекта (на стадии выполнения).

Подавляющее большинство остальных классов являются наследниками класса **TObject** и, далее, класса **TPersistent**.

В рассматриваемых ниже примерах все классы-потомки ведут происхождение непосредственно от базового класса **TObject**, прежде всего потому, что эти примеры являются примерами простейшего типа и не требуют возможностей, выходящих за пределы базового класса.

Организация простейшего класса.

Рассмотрим организацию простейшего класса, являющегося потомком базового класса **TObject**.




Рассмотрение начнём с создания класса **TOperation**. Данный класс наделим единственным методом (**summ**). Этот метод будет возвращать сумму аргументов **(a,b)**, подаваемых на его вход:

Листинг 2.1.

```
Unit Unit_operation;
interface
 type
```

```
  TOperation = class(TObject)
   Function summ(a,b:integer):integer;
  end;
implementation
Function TOperation.summ(a,b:integer):integer;
  begin
    summ:= a+b;
  end;
end.
```

Кратко прокомментируем приведённый выше код:

– наименование модуля - "**Unit_operation**", и он сохраняется в файле с именем "**Unit_operation.pas**";

– в секции **type** раздела **interface** модуля прописано определение класса **TOperation**;

– определение класса начинается заголовком "**TOperation= class(TObject)**" и заканчивается словом "**end;**";

– в классе **TOperation** определён метод - "**summ(a,b:integer)**". Данный метод - единственный метод класса **TOperation**;

– следует обратить внимание на различие заголовка метода в разделах **interface** и **implementation**. В первом случае имя класса не приводится, во втором - заголовок состоит из двух частей, разделённых точкой "**Function TOperation.summ (a,b:integer): integer;**". В первой части фигурирует имя класса, во второй - имя метода с указанием параметров, подаваемых на вход этого метода;

– в разделе **implementation** модуля приводится расшифровка метода **summ** класса **TOperation**;

– в определении класса **TOperation** не задействован конструктор класса. Из последующего изложения станет ясно, что подобный упрощённый и облегчённый вариант организации класса возможен только в тривиальных случаях.

Использование простейшего класса.

Для использования возможностей организованного нами класса **TOperation** необходимо создать небольшой **Lazarus**-проект. В состав этого проекта должны войти следующие файлы:

– головной файл проекта (файл с расширением **dpr**), в котором указано, какие файлы из числа тех, с которыми напрямую работает пользователь (файлы с расширением **pas**), входят в состав проекта. Имя головному файлу проекта присваивает пользователь, когда сохраняет проект. В рассматриваемом нами примере присвоим проекту имя **Class_definition**. Исполняемому файлу проекта, соответственно, при сборке проекта будет присвоено имя "**Class_definition.exe**";

– файл, в котором располагается определение класса **TOperation**. Имя этого файла - "**Unit_operation.pas**";

– файл, в котором располагаются средства управления возможностями класса **TOperation**. Имя этого файла - "**Unit_Program.pas**";

– остальные файлы проекта образуются без какого-либо прямого участия пользователя.

Рассмотрим структуру перечисленных выше файлов.

Поскольку и файл "**Unit_operation.pas**", и комментарии к нему приведены выше, осталось рассмотреть два других файла. Рассмотрение начнём с файла "**Class_definition.dpr**":

Листинг 2.2.

```
program Class_definition;
uses
  Forms,
  Unit_Program in 'Unit_Program.pas' {Form_Program},
  Unit_Operation in 'Unit_Operation.pas';
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm_Program, Form_Program);
  Application.Run;
End.
```

Краткий комментарий:

– в состав проекта входят модули **Unit_Program** и **Unit_operation**;

– модуль **Unit_operation** имеет простейшую структуру и не связан с какими-либо формами;

– модуль **Unit_Program** имеет более сложную структуру и включает в свой состав форму с именем **Form_Program**.

Перейдём к рассмотрению структуры файла "**Unit_Program.pas**". В данном файле располагается модуль **Unit_Program**. Код этого модуля с некоторыми сокращениями приведен ниже.

Листинг 2.3.

```
Unit Unit_Program;
interface
uses ..., Unit_operation;

type

  TForm_Program = class(TForm)
   <Определения полей >
   <Определения методов >
  end;

var Form_Program: TForm_Program;
    Sm: TOperation;

implementation
{$R *.dfm}

procedure TForm_Program.Panel1Click(Sender: TObject);
var x,y,r: Integer;
begin
  x:= StrToInt(Edit1.Text);
  y:= StrToInt(Edit2.Text);
  r:= Sm.Summ(x,y);
  Label1.Caption:= IntToStr(r);
```

```
end;

end.
```

Комментарий к приведённому выше коду:

– модуль состоит из двух разделов **interface** и **implementation**;

– в разделе **interface** модуля располагаются секции **uses**, **type** и **var**;

– к перечню модулей в секции **uses** следует добавить модуль **Unit_operation** (остальные модули к перечню подключаются автоматически);

– в секции **type** дано определение класса **TForm_Program**. Это определение сформировано автоматически, без участия пользователя. Пользователь только дал имя форме: **Form_Program**. Всё остальное **Lazarus** выполнила автоматически. На форме расположены два текстовых окна **TEdit** (для ввода слагаемых метода **Summ**), панель **TPanel** (для запуска процедуры **Panel1Click**), метка **TLabel** (для публикации результата применения метода **Summ**). Единственный метод класса **TForm_Program** реализован в виде процедуры **Panel1Click** (для запуска метода **Summ** и публикации результата его применения);

– в секции **var** определены переменная **Form_Program** класса **TForm_Program** и переменная **Sm** класса **TOperation**. Переменная **Sm** играет ключевую роль. Именно в ней будут храниться экземпляры класса **TOperation**;

– в разделе **implementation** модуля располагается расшифровка метода **Panel1Click**, объявленного ранее в секции **type** раздела **interface**;

– локальные переменные ("**var x,y,r: Integer;**") используются при применении метода **summ** класса **TOperation**;

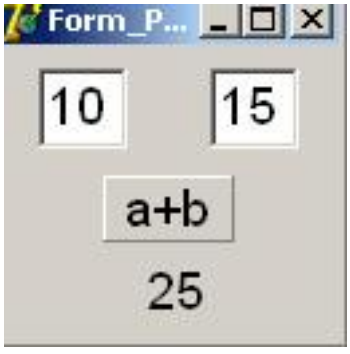
– код "**x:= StrToInt(Edit1.Text);**" позволяет считать содержимое текстового окна и присвоить считанное значение (после трансформации его в тип **integer**) целочисленной переменной **x**;

– код "**r:= Sm.Summ(x,y);**" обеспечивает обращение к методу **summ** класса **TOperation**. Результат суммирования (**x+y**) присваивается целочисленной переменной **r**;

– код "**Label1.Caption:= IntToStr(r);**" обеспечивает визуализацию значения переменной **r**. В этой переменной содержится результат суммирования (**x+y**).

Запуск подготовленного проекта.

Запуск откомпилированного проекта на исполнение (результат компиляции находится в файле "**Class_definition.exe**") приводит к следующему результату:




Копия фрагмента экрана выполнена после щелчка по панели **Panel1**. Данная панель имеет заголовок (**Caption**), которому задано значение "**a+b**". Нанесённый щелчок привёл к выполнению кода процедуры **Panel1Click**. В результате переменная **x** получает значение **10** (считано из окна **Edit1** и преобразовано к целочисленному типу данных), переменная "**y**" получает значение **15** (считано из окна **Edit2**).

Далее, выполняется код "**Sm.Summ(x,y);**". Это означает, что выполняется метод **Summ** класса **TOperation**. Выполнение данного метода осуществляется экземпляром класса **TOperation**. Экземпляр (объект) представляет собою значение объектной

переменной **Sm**. Поскольку к этому времени переменным **x** и **y** уже присвоены конкретные значения, выполняемый код имеет вид: **Sm.Summ(10,15)**.

В результате выполнения метода **Summ** класса **TOperation** выполняется суммирование аргументов метода, после чего функция **Summ** возвращает рассчитанное значение суммы **25**, которое присваивается переменной **r**.

Значение переменной **r** конвертируется в строку, после чего результат конвертирования становится заголовком "**25**" метки **Label1**.

## Задания для самостоятельной работы

1. Модернизировать класс **TOperation**, добавив ещё один метод (**Diff(a,b)**), который бы возвращал разность поданных на его вход аргументов. В модуль **Unit_Program** внести дополнения, позволяющие использовать метод **Diff**.

2. Создать новый класс **TOper_mult**, в котором определить метод (**Mult(a,b,c)**), возвращающий произведение поданных на его вход аргументов. Создать модуль, позволяющий использовать метод **Mult**.

3. Создать новый проект, перенеся определение класса **TOper_mult** в модуль формы. Обеспечить возможность использования метода **Mult**.

***ЛАБОРАТОРНОЕ ЗАНЯТИЕ № 11-13.***

*Тема:* Реализация базовых понятий ООП

**Реализация базовых понятий ООП в Lazarus: наследование.**

Как известно, к числу базовых понятий ООП относятся такие понятия как инкапсуляция, наследование и полиморфизм. Рассмотрение этих базовых понятий начнём с рассмотрения понятия наследования.

Принцип наследования реализован в системе программирования **Lazarus** следующим образом:

– любой класс **Lazarus** может рассматриваться как класс-родитель, то есть как класс, способный порождать производные классы, именуемые классами-потомками;

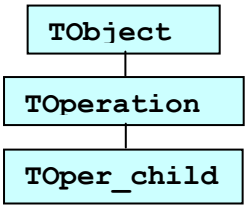
– класс-потомок может вести своё происхождение от любого класса-родителя, но только одного родителя. Другими словами, множественное наследование (несколько классов-родителей) для одного класса-потомка системой программирования **Lazarus** не поддерживается;

– класс-потомок наследует все методы, поля и свойства класса-родителя. Наследование классом-потомком методов, полей и свойств класса-родителя означает, что класс-наследник может пользоваться методами, полями и свойствами класса-родителя без каких-либо дополнительных определений;

– определение в классе-наследнике каких-либо методов, полей или свойств из числа тех, которыми обладает класс-родитель, означает их переопределение. Осуществлённое переопределение блокирует/экранирует переопределённые методы, поля или свойства класса-родителя. После переопределения класс-потомок может использовать исключительно собственные методы, поля или свойства, которые заменили соответствующие родительские методы, поля или свойства;

– наследование методов, полей или свойств класса-родителя реализуемо не только для прямого класса-потомка, но и для потомков потомка.

Чтобы лучше разобраться, рассмотрим конкретный пример. Приступим к построению класса-потомка. В качестве родителя класса-потомка выберем ранее построенный класс **TOperation**. Классу-потомку подберём имя **TOper_child**:




Организация простейшего класса-потомка.

Класс-потомок **TOper_child** наделим единственным собственным методом (**Diff**), возвращающим разность аргументов **(a,b)**, подаваемых на его вход.

Модуль, в котором будут определены класс-родитель и класс-потомок, можно организовать следующим образом:

Листинг 3.1.

```
Unit Unit_operation;
interface
 type
```

```
  TOperation = class(TObject)
   Function summ(a,b:integer):integer;
  end;
  TOper_child = class(TOperation)
   Function diff(a,b:integer):integer;
  end;
implementation
Function TOperation.summ(a,b:integer):integer;
   begin
     summ:= a+b;
   end;
Function TOper_child.diff(a,b:integer):integer;
   begin
     diff:= a-b;
   end;
end.
```

Сопроводим приведённый выше код кратким комментарием:

– в секции **type** раздела **interface** модуля прописаны определения двух классов **TOperation** и **TOper_child**;

– определение каждого класса заканчивается словом "**end;**";

– в классе **TOperation** определён метод - "**summ(a,b:integer)**". Данный метод - единственный метод класса **TOperation**;

– в классе **TOper_child** явно также задан всего лишь один метод - "**diff (a,b:integer)**", однако, ещё один метод, метод **summ**, наследуется у класса-родителя **TOperation**. Далее будет показано, как методом **summ** класс-потомок может воспользоваться и как этот метод в классе-потомке может быть переопределён;

– в разделе **implementation** модуля приводятся расшифровки методов класса-родителя и класса-потомка.

Использование класса-потомка.

Для использования возможностей организованного нами класса-потомка **TOper_child** необходимо создать соответствующий **Lazarus**-проект. В данном проекте следует предусмотреть наличие следующих компонентов:

– однострочные текстовые окна для задания операндов операций сложения и вычитания (**a,b**): (**Edit1** и **Edit2**);

– панели (**Panel1** и **Panel2**) для запуска процедур **Panel1Click** и **Panel2Click**;

– метки (**Label1** и **Label2**) для публикации результатов применения методов **Summ** и **Diff**.

Поскольку всё, что имеет отношение к классу **TOperation**, в модуле **Unit_Program** не меняется, к коду, расположенному в модуле, остаётся добавить только те строчки, которые имеют отношение к классу **TOper_child**:

– во-первых, в секции **var** раздела **interface** модуля следует объявить переменную **Df** класса **TOper_child**, в которой будут храниться экземпляры (объекты) этого класса;

Листинг 3.2.

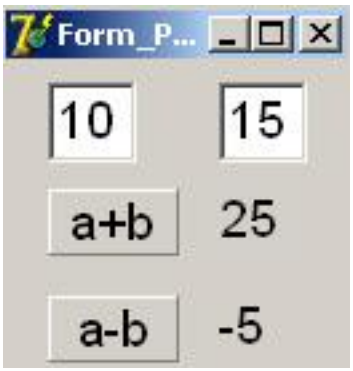
```
var ...
   Df: TOper_child;
```

– во-вторых, в разделе **implementation** модуля следует расположить код процедуры, обеспечивающей выполнение метода **Diff** класса **TOper_child**.

Листинг 3.3.

```
procedure TForm1.Panel2Click (Sender: TObject);
var x,y,t: Integer;
begin
  x:= StrToInt(Edit1.Text);
  y:= StrToInt(Edit2.Text);
  t:= Df.Diff(x,y);
  Label2.Caption:= IntToStr(t);
end;
```

Запуск откомпилированного проекта на исполнение приводит к следующему результату:




Копия фрагмента экрана выполнена после щелчков по панелям **Panel1** и **Panel2**, имеющим заголовки "**a+b**" и "**a-b**". Нанесение щелчков инициировало выполнение кода процедур **Panel1Click** и **Panel2Click**. Результаты выполнения процедур отображены на метках **Label1** и **Label2** соответственно.

Использование классом-потомком метода класса-родителя.

Для использования классом-потомком **TOper_child** метода класса-родителя **TOperation** в существующий **Lazarus**-проект необходимо внести небольшое изменение. В процедуре **Panel1Click** строку "**r:= Sm.Summ(x,y);**" следует заменить строкой "**r:= Df.Summ(x,y);**".

Внесённое небольшое изменение носит принципиальный характер: если ранее метод **Summ** использовался экземпляром класса-родоначальника (т.е. класса **TOperation**) этого метода ("**Sm.Summ(x,y);**"), теперь метод **Summ** используется экземпляром класса-потомка (**TOper_child**): "**Df.Summ(x,y);**".

Такое использование метода класса-родителя называется наследованием метода родителя классом-потомком.

Запуск проекта на исполнение не приведёт к каким-либо визуальным изменением. Метод **Summ** теперь будет выполнен не экземпляром класса **TOperation**, а экземпляром класса **TOper_child**. А возвращенное функцией **Summ** рассчитанное значение окажется тем же самым и равным **25**.

Переопределение классом-потомком метода класса-родителя.

Для переопределения метода класса-родителя в классе-потомке следует разместить определение, которое бы каким-то образом отличалось от исходного родительского определения (не имеет смысл переопределять метод, не внося каких-либо отличий).

Предположим, мы решили переопределить метод **Summ** класса-родителя в классе-потомке таким образом, чтобы вычислялась не сумма слагаемых, а квадрат этой суммы.

Для реализации этой цели в модуль **Unit_operation** следует внести следующее изменение:

Листинг 3.4.

```
...
  TOper_child = class(TOperation)
   Function diff(a,b:integer):integer;
   Function summ(a,b:integer):integer;
  end;
implementation
...
Function TOper_child.summ(a,b:integer):integer;
   begin
     summ:= (a+b)*(a+b);
   end;
```

Краткий комментарий к внесённым изменениям:

– в заголовочной структуре класса **TOper_child** приводится заголовок переопределяемого метода **summ** ("**Function summ(a,b:integer):integer;"**);

– в разделе **implementation** модуля **Unit_operation** приводится расшифровка переопределённого метода суммирования: "**TOper_child.summ (a,b:integer): integer;**". На этот раз возвращаемое функцией **summ** значение определяется как квадрат суммы аргументов: "**summ:= (a+b)*(a+b);**".

Запуск проекта на исполнение приведёт к визуализации результата применения переопределённого метода **Summ**. На экране будет высвечено (аргументы те же самые: **10** и **15**) значение **225**.

## Задания для самостоятельной работы

1. Создать класс-потомок **TOper_mult** класса **TOperation** в котором определить метод (**Mult(a,b)**), возвращающий произведение поданных на его вход аргументов. Создать модуль, позволяющий использовать методы родительского и дочернего классов.

2. Создать класс-потомок **TOper_quat** класса **TOper_mult** в котором определить метод (**Quat(a,b)**), возвращающий сумму квадратов поданных на его вход аргументов. Проверить наследование классом **TOper_quat** методов классов **TOper_mult и TOperation**.

3. Перенести определения классов **TOper_quat** и **TOper_mult** в другой модуль. Проверить наследование классами **TOper_quat** и **TOper_mult** метода класса **TOperation**.

## Реализация базовых понятий ООП в Lazarus: инкапсуляция.

Рассмотрение начнём с разбора понятия инкапсуляции и того, как это понятие реализовано в среде программирования **Lazarus**.

Под инкапсуляцией принято понимать сокрытие, защиту данных. Инкапсуляция обеспечивает ограничение доступа к инкапсулируемым данным.

Понятие инкапсуляции может быть применено как к классу, так и к модулю. Это означает, что мы можем как открыть, так и закрыть доступ к данным, расположенным как в классе, так и в модуле.

Рассмотрим на конкретном примере, как это закрытие/открытие данных может быть реализовано.

В § 2 рассмотрен простейший класс (**TOperation**), наделённый единственным методом (**summ**), методом суммирования поданных на вход аргументов (**a,b**). По умолчанию, все методы определяются как **public**-методы, поэтому использование метода **summ** не встретило каких-либо трудностей.

Внесём в листинг 2.1 коррективы, а именно, внедрим в структуру класса **TOperation** секцию **private**, и именно в эту секцию перенесём заголовок метода **summ**. Результат этой корректировки отображен в листинге 4.1.

Листинг 4.1.

```
Unit Unit_operation;
interface
 type
  TOperation = class(TObject)
   private
   Function summ(a,b:integer):integer;
  end;
. . .
end.
```

В листинге 2.3 ничего менять не будем. Попробуем из **Lazarus** запустить данный проект на исполнение. При компиляции модуля **Unit_Program** будет выдано сообщение об ошибке:

**[Error] Unit_Program.pas(30): Undeclared identifier: 'Summ'**

Прокомментируем эту ошибку:

– ошибка допущена в 30-ой строке кода модуля "**Unit_Program.pas**";

– 30-ая строка кода имеет вид: "**r:= Sm.Summ(x,y);**";

– в сообщении сказано, что идентификатор **Summ** не объявлен.

Итак, объявление метода **Summ** в разделе **private** модуля **Unit_operation** привело к тому, что в другом модуле (**Unit_Program**) об этом объявлении теперь ничего неизвестно. Естественно, не будучи уведомлен об этом объявлении, компилятор выдаёт сообщение об ошибке.

Если в структуру класса **TOperation** вместо секции **private** внедрить секцию **protected**, результат окажется тем же: будет выдано сообщение о той же ошибке.

Попробуем теперь воспользоваться защищенным методом не из другого модуля, а из того же самого.

В § 3 (Реализация базовых понятий ООП в **Lazarus**: наследование) рассмотрены родительский и дочерний классы **TOperation** (наделён единственным методом **summ**) и **TOper_child** соответственно. Эти методы (листинг 3.1) определены как **public**-методы.

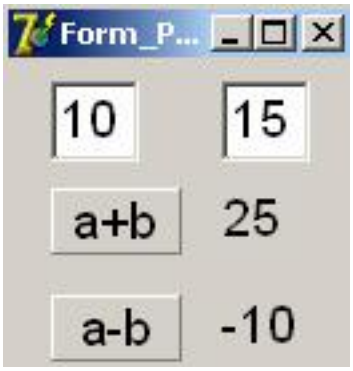
Внесём в листинг 3.1 коррективы, а именно, внедрим в структуру класса **TOperation** секцию **private** и внесём в эту секцию заголовок нового метода **dubl**, который будет удваивать значения поданных на его вход аргументов. Результат этой корректировки отображен в листинге 4.2.

Листинг 4.2.

```
Unit Unit_operation;
interface
 type
  TOperation = class(TObject)
```

```
  Function summ(a,b: integer): integer;
  private
  Procedure dubl(var a,b: integer);
 end;

 TOper_child = class(TOperation)
  Function diff(a,b:integer):integer;
 end;

implementation

Function TOperation.summ(a,b: integer): integer;
  begin
    summ:= a+b;
  end;

Procedure TOperation.dubl(var a,b: integer);
  begin
    a:= 2*a; b:= 2*b;
  end;

Function TOper_child.diff(a,b:integer):integer;
  begin
    dubl(a,b);
    diff:= a-b;
  end;
end.
```

В модуль "**Unit_Program.pas**" изменений вносить не будем. Запуская проект на исполнение, получаем:




Краткий комментарий к полученному результату:

– копия фрагмента экрана выполнена после щелчков по панелям озаглавленных как "**a+b**" и "**a-b**";

– нанесение щелчка по панели с заголовком "**a-b**" привело к запуску процедуры **Panel2Click**;

– код "**t:= Df.Diff(x,y);**" процедуры **Panel2Click** обеспечивает запуск метода **Diff** с параметрами **10** и **15**: "**Df.Diff(10,15);**";

– выполнение метода **Diff** осуществляется в два этапа: сначала запускается метод "**dubl(10,15)**", затем запускается собственный метод дочернего класса "**diff:= 20-30;**";

– на экране будет высвечено значение "**-10**".

После получения приведённого выше положительного результата осталось проверить, можно ли воспользоваться защищенным методом из другого модуля.

Рассмотренный выше модуль **Unit_operation** разобьём на два модуля: **Unit_oper_1** и **Unit_oper_2**.

В модуле **Unit_oper_1** разместим определение класса- родителя **TOperation**.

Листинг 4.3.

```
Unit Unit_oper_1;

interface
 type
  TOperation = class(TObject)
   Function summ(a,b: integer): integer;
   private
   Procedure dubl(var a,b: integer);
  end;

implementation

Function TOperation.summ(a,b: integer): integer;
   Begin summ:= a+b; end;

Procedure TOperation.dubl(var a,b: integer);
   Begin a:= 2*a; b:= 2*b; end;
end.
```

В модуле: **Unit_oper_2** разместим определение класса-потомка **TOper_child**.

Листинг 4.4.

```
unit Unit_oper_2;

interface
uses Unit_oper_1;
type
  TOper_child = class(TOperation)
   Function diff(a,b:integer):integer;
  end;

implementation
Function TOper_child.diff(a,b:integer):integer;
   begin
     dubl(a,b);
     diff:= a-b;
   end;
```

```
end.
```

В модуль **Unit_Program** следует внести единственное исправление: код "**uses Unit_operation;**" следует заменить кодом "**uses Unit_oper_1,Unit_oper_2;**".

Попробуем запустить данный проект на исполнение из среды **Lazarus**. При компиляции модуля **Unit_oper_2** будет выдано сообщение об ошибке:

**[Error] Unit_oper_2.pas(13): Undeclared identifier: 'dubl'**

Компилятору не нравится конструкция "**dubl(a,b);**", так как она определена в другом модуле (**Unit_oper_1**) со спецификатором **private**.

Изменим спецификатор: вместо **private** подставим **protected**. Компиляция модуля **Unit_oper_2** проходит без замечаний, на экране высвечивается знакомый результат: "**-10**".

Итак, использование конструкций какого-либо класса из другого модуля возможно только в случае, когда выполнены следующие два условия:

– используемая конструкция класса определена со спецификатором **protected**;

– использование конструкции, помеченной спецификатором **protected**, осуществляется из другого модуля при расшифровке какого-либо метода класса-потомка.

## Задания для самостоятельной работы

1. Создать класс-потомок **TOper_mult** класса **TOperation** в котором определить метод (**Mult(a,b)**), возвращающий произведение поданных на его вход аргументов. Определение класса-потомка **TOper_mult** разместить в другом модуле. Создать модуль, позволяющий использовать методы как родительского, так и дочернего классов.

2. Добавить к методам класса **TOperation** метод, позволяющий возвращать два параметра: сумму и произведение подаваемых на вход метода параметров. Пометить данный метод спецификатором доступа **protected**. Использовать этот метод при определении нового метода класса-потомка **TOper_mult**. Например, метода, возвращающего сумму квадратов аргументов, подаваемых на его вход.

## Реализация базовых понятий ООП в Lazarus: полиморфизм.

Полиморфизм, наряду с такими понятиями как наследование и инкапсуляция, относится к числу базовых понятий ООП.

Полиморфизм представляет собою способность экземпляра того или иного класса выступать от имени класса-потомка, заимствуя у класса-потомка интерпретацию своих методов, полей и свойств.

Свойство полиморфизма разрешает экземпляру класса-родителя принимать значение экземпляра класса-потомка. В результате все методы, поля и свойства экземпляра класса-родителя становятся копиями соответствующих методов, полей и свойств экземпляра класса-потомка.

Что означает сказанное выше, рассмотрим на конкретном примере.

Пример. Предположим, класс-родитель владеет методом, позволяющим возводить подаваемый на его вход аргумент, в квадрат. Предположим далее что, класс-потомок владеет методом (с тем же наименованием), позволяющим возводить подаваемый на его вход аргумент, в куб.

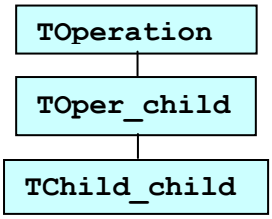
Выполним присваивание: **<объект-родитель> := <объект-потомок>**.

После этого присваивания метод **<объекта-родителя>** будет выполнять операцию возведения подаваемого на его вход аргумента в куб.

Способность **<объекта-родителя>** возводить аргумент в квадрат будет фактически стерта из памяти, **<объект-родитель>** забудет, как эта операция выполняется.

Возводить аргумент в квадрат без посторонней помощи **<объект-родитель>** более не сможет.

По завершении введения, перейдём к рассмотрению конкретного примера. Определим метод **Summ** для трёх классов, образующих следующую иерархическую структуру:




Модуль **Unit_operation**, в котором определены все три класса, представлен ниже.

Листинг 7.1.

```
Unit Unit_operation;
interface
 type
  TOperation = class(TObject)
    Function summ(a,b: integer): integer; virtual;
  end;

  TOper_child = class(TOperation)
   Function summ(a,b: integer): integer; override;
  end;

  TChild_child = class(TOper_child)
   Function summ(a,b: integer): integer; override;
  end;

implementation

Function TOperation.summ(a,b: integer): integer;
   Begin  summ:= a+b;  end;

Function TOper_child.summ(a,b:integer):integer;
   Begin  summ:= (a+b)*(a+b);  end;

Function TChild_child.summ(a,b:integer):integer;
   Begin  summ:= (a+b)*(a+b)*(a+b);  end;

End.
```

Комментарий к приведённому выше коду:

– метод **summ** класса-родителя **TOperation** в строке заголовке помечен ключевым словом "**virtual**": "**Function summ(a,b:integer): integer; virtual;**";

– отметим, что техника полиморфизма применима только для динамических методов;

– метод **summ** классов-потомков в строке заголовке помечен ключевым словом "**override**": "**Function summ(a,b: integer): integer; override;**";

– обратим внимание на то, что классы-потомки, используемые при реализации полиморфизма, обязательно помечаются ключевым словом "**override**";

– конкретная схема реализации полиморфизма определяется кодом модуля **Unit_Program**, отраженном в листинге 7.2:

Листинг 7.2.

```
Unit Unit_Program;
interface
. . .
implementation
{$R *.dfm}
uses Unit_operation;
var
 Op0,Op1: TOperation;
    Op2: TOper_child;
    Op3: TChild_child;

procedure TForm_Program.Panel1Click(Sender: TObject);
var x,y,r: Integer;
begin
  x:= StrToInt(Edit1.Text);
  y:= StrToInt(Edit2.Text);

  Op1:= TOperation.Create;
  Op0:= Op1;
  Op2:= TOper_child.Create;
  Op3:= TChild_child.Create;

  r:= Op1.Summ(x,y);
  Label1.Caption:= IntToStr(r);

  Op1:= Op2;
  r:= Op1.Summ(x,y);
  Label2.Caption:= IntToStr(r);

  Op1:= Op3;
  r:= Op1.Summ(x,y);
  Label3.Caption:= IntToStr(r);

  Op1:= Op0;
  r:= Op1.Summ(x,y);
  Label0.Caption:= IntToStr(r);
end;
end.
```

Прокомментируем приведённый выше код:

– созданный экземпляр класса **TOperation** сохраняется в объектных переменных **Op1** и **Op0**;

– вторая переменная **Op0** потребовалась, чтобы запомнить первоначальное значение переменной **Op1**;

– созданный экземпляр класса **TOper_child** сохраняется в объектной переменной **Op2**;

– созданный экземпляр класса **TChild_child** сохраняется в объектной переменной **Op3**;

– применение полиморфизма начинается с присваивания "**Op1:= Op2;**", которое превращает экземпляр класса **TOperation** в экземпляр класса **TOper_child**. С этого момента экземпляр **Op1** начинает вести себя как экземпляр **Op2**. Это проявляется в том, что метод "**Summ(x,y)**" экземпляра **Op1** перестаёт работать по схеме "**summ:= a+b;**" и начинает работать по схеме "**summ:= (a+b)*(a+b);**";

– применение полиморфизма продолжается выполнением присваивания "**Op1:= Op3**", которое превращает модифицированный экземпляр класса **TOperation** теперь уже в экземпляр класса **TChild_child**. С этого момента экземпляр **Op1** начинает вести себя как экземпляр **Op3**. Это проявляется в том, что метод "**Summ(x,y)**" экземпляра **Op1** перестаёт работать по схеме "**summ:= (a+b)*(a+b);**" и начинает работать по схеме "**summ:= (a+b)*(a+b)*(a+b);**";

– применение полиморфизма завершается выполнением присваивания "**Op1:= Op0**", которое превращает дважды модифицированный экземпляр класса **TOperation** в не модифицированный экземпляр класса **TOperation**. С этого момента экземпляр **Op1** начинает вести себя так, как это было до первого из присваиваний "**Op1:= Op2;**". Это проявляется в том, что метод "**Summ(x,y)**" экземпляра **Op1** вновь начинает работать по схеме "**summ:= (a+b);**".

## Задания для самостоятельной работы

1. Создать класс **TOper_summ**, в котором определить динамический (**virtual**) метод **summ**, возвращающий сумму подаваемых на его вход 2-х аргументов.

2. Создать класс-потомок **TSumm_summ** класса **TOper_summ**, в котором определить метод **summ**, возвращающий сумму квадратов подаваемых на его вход 2-х аргументов.

3. Реализовать полиморфизм для экземпляра класса **TOper_summ**.

***ЛАБОРАТОРНОЕ ЗАНЯТИЕ № 14-16.***

*Тема:* Реализация базовых понятий ООП

## Перегрузка в ООП.

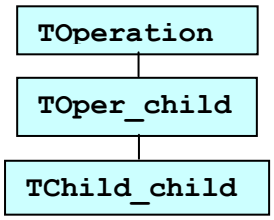
В ООП существует приём, именуемый **перегрузкой** (**Overload**). Данный приём позволяет использовать два и более метода с одинаковым именем, но с различными наборами параметров (различными сигнатурами).

Чтобы воспользоваться перегрузкой, каждый из перегружаемых методов следует пометить ключевым словом **Overload**.

Поскольку методы реализуются как в виде процедур, так и виде функций, часто говорят и о перегрузке функций. Следует подчеркнуть, что и в одном, и в другом случае речь идёт об одном и том же приёме - **перегрузке** подпрограммы. Перегрузка позволяет, не меняя имени метода, выполнять различные подпрограммы.

Если и в классе-родителе, и в классе-потомке, определен метод с одним и тем же именем, **без перегрузки** родительский метод в классе-потомке будет **переопределён**. Это означает, что методом родительского класса в классе-потомке невозможно воспользоваться. Использование перегрузки позволяет с этой задачей справиться.

Рассмотрим применение перегрузки метода на конкретном примере. Допустим, перегружаемым методом будет метод **Summ**. Определим этот метод для трёх классов, образующих следующую иерархическую структуру:




Модуль **Unit_operation**, в котором будут определены все три класса, можно организовать следующим образом:

Листинг 5.1.

```
Unit Unit_operation;

interface
 type
  TOperation = class(TObject)
   Function summ(a,b: integer): integer; overload;
  end;

  TOper_child = class(TOperation)
   Function diff(a,b:integer):integer;
   Function summ(a,b,c: integer): integer; overload;
  end;

  TChild_child = class(TOper_child)
   Function mult(a,b:integer):integer;
   Function summ(a,b,c,d: integer): integer; overload;
  end;
```

```
implementation

Function TOperation.summ(a,b: integer): integer;
  Begin  summ:= a+b;  end;

Function TOper_child.diff(a,b:integer):integer;
  Begin  diff:= a-b;  end;

Function TOper_child.summ(a,b,c:integer):integer;
  begin  summ:= a+b+c;  end;

Function TChild_child.mult(a,b:integer):integer;
  begin  mult:= a*b;  end;

Function TChild_child.summ(a,b,c,d:integer):integer;
  begin  summ:= a+b+c+d;  end;
end.
```

Краткий комментарий к приведённому выше коду:

– в каждом из трёх классов **TOperation**, **TOper_child** и **TChild_child** определён метод **summ**, однако, реализован он в этих классах по-разному;

– в классе **TOperation**: "**summ:=a+b;**";

– в классе **TOper_child**: "**summ:=a+b+c;**";

– в классе **TChild_child**: "**summ:=a+b+c+d;**";

– объявление метода **summ** в каждом из классов помечено словом **overload**:

**Function summ(a,b: integer): integer; overload;**

**Function summ(a,b,c: integer): integer; overload;**

**Function summ(a,b,c,d: integer): integer; overload;**

– классы **TOper_child** и **TChild_child** помимо метода **summ** имеют по одному собственному методу. Речь идёт о методах **diff** (класс **TOper_child**) и **mult** (класс **TChild_child**).

Приступим к рассмотрению следующего вопроса: каким образом перегрузка метода **summ** может быть использована.

В модуль **Unit_Program** внесём следующие коррективы. Введём в рассмотрение переменные объектного типа (**Sm**, **Df**, **Ml**), позволяющие воспользоваться перегружаемым методом **summ**:

Листинг 5.2.

```
Uses Unit_operation;
var
    Sm: TOperation;
    Df: TOper_child;
    Ml: TChild_child;
```

Проверим, какие варианты перегрузки возможны.

Родительский класс **TOperation** позволяет выполнить обращение только к одному варианту метода **summ** (к своему собственному):

Листинг 5.3.

```
R1:= Sm.Summ(x,y);
```

Класс **TOper_child** позволяет выполнить обращение уже к двум вариантам метода **summ** (к методу родителя и своему собственному методу):

Листинг 5.4.

```
R1:= Df.Summ(x,y);
R2:= Df.Summ(x,y,z);
```

Класс **TChild_child** позволяет выполнить обращение ко всем трём вариантам метода (к методам родителя родителя, просто родителя и своему собственному методу):

Листинг 5.5.

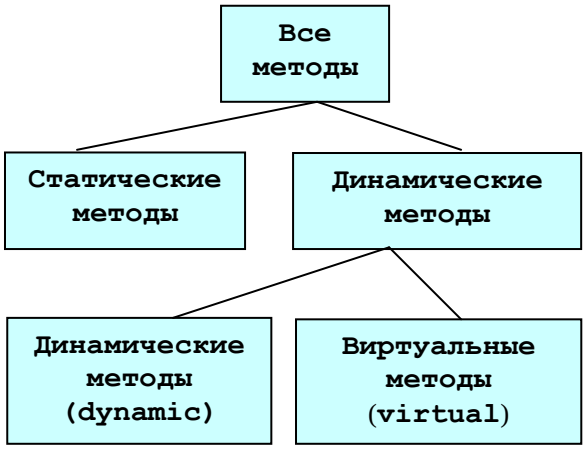
```
R1:= Ml.Summ(x,y);
R2:= Ml.Summ(x,y,z);
R3:= Ml.Summ(x,y,z,t);
```

## Задания для самостоятельной работы

1. Создать класс **TOper_mult**, в котором определить два варианта перегружаемого метода **mult** с числом параметров 2 и 3.
2. Создать класс-потомок **TMult_mult** класса **TOper_mult**, в котором доопределить ещё один перегружаемый метод **Mult** с 4 параметрами. Определение класса-потомка **TMult_mult** разместить в другом модуле.

## Виртуальные и динамические методы в ООП и Lazarus.

В ООП существуют принципиально различные подходы к определению процедур и функций. Данные подходы иллюстрируются приводимой ниже диаграммой.




До сих пор, в предыдущих параграфах, велось рассмотрение исключительно статических функций и процедур. Статус статических этим процедурам и функциям не присваивался, поскольку: все процедуры и функции считаются статическими без каких-либо объявлений, т.е. по умолчанию.

Отличия между статическими и нестатическими процедурами и функциями значительны. Нестатические процедуры и функции бывают двух типов: динамические (обозначаются ключевым словом **dynamic**) и виртуальные (обозначаются ключевым словом **virtual**).

В простейших случаях динамические и виртуальные методы практически не различимы. Следует отметить, что на практике существенно чаще используются виртуальные методы. По этой причине в рассматриваемых ниже примерах предпочтение будет отдаваться виртуальным методам.

Принципиальное различие между статическими и динамическими методами заключается в следующем.

Статический метод полностью определяется уже на стадии конструирования проекта. К моменту компиляции проекта известны все обращения к статическому методу, известны условия этих обращений, словом, известно всё, что только может быть известным.

Если в обращениях к статическому методу присутствуют ошибки, эти ошибки выявляются на стадиях компиляции и сборки проекта. Это означает, что известно, какого типа ошибка, и в какой строке эта ошибка находится.

Динамический метод конструируется на стадии выполнения откомпилированного и собранного проекта. Вследствие этого, фактический адрес динамического метода в принципе не может быть известен на стадии компиляции проекта. Это означает, что тип ошибок на стадии выполнения собранного проекта не детектируется в принципе, поэтому устранение такого рода ошибок максимально усложняется с самого начала.

В ООП, применительно к статическим и динамическим методам, вводятся понятия позднего и раннего связывания. О раннем связывании принято говорить, когда речь идёт о статических методах. Позднее связывание упоминают, когда речь идёт о методах динамических и виртуальных.

По завершении краткого введения переходим к рассмотрению конкретных примеров.

Построим класс **TOperation**, в котором определим единственный виртуальный метод **summ**, позволяющий суммировать подаваемые на его вход аргументы.

Модуль **Unit_operation**, в котором определим наш класс, представлен ниже.

Листинг 6.1.

```
Unit Unit_operation;

interface
 type
  TOperation = class(TObject)
   public
    Function summ(a,b: integer): integer; virtual;
  end;

implementation

Function TOperation.summ(a,b: integer): integer;
   begin
     summ:= a+b;
   end;
end.
```

Краткий комментарий:

– единственное отличие данного модуля от модуля представленного листингом 2.1, наличие слова "**virtual**" в строке-заголовке метода: "**Function summ(a,b: integer): integer; virtual;**".

Что касается модуля **Unit_Program**, здесь также присутствует отличие принципиального плана:

Листинг 6.2.

```
Unit Unit_Program;

interface

uses ..., Unit_operation;

type

  TForm_Program = class(TForm)
   <Определения полей >
   <Определения методов >
  end;

var Form_Program: TForm_Program;
    Sm: TOperation;

implementation
{$R *.dfm}

procedure TForm_Program.Panel1Click(Sender: TObject);
var x,y,r: Integer;
begin
  x:= StrToInt(Edit1.Text);
  y:= StrToInt(Edit2.Text);

  Sm:= TOperation.Create;

  r:= Sm.Summ(x,y);
  Label1.Caption:= IntToStr(r);
end;

end.
```

Прокомментируем приведённый выше код:

– ключевое отличие от кода, приведённого в листинге 2.3, присутствие строки, обеспечивающей конструирование экземпляра класса **TOperation**: "**Sm:= TOperation.Create;**";

– сконструированный экземпляр класса **TOperation** сохраняется в переменной объектного типа **Sm**.

Ещё раз обратим внимание на ключевую строку приведённого кода:

**"Sm:= TOperation.Create;"**

Метод **Create**, обеспечивающий создание экземпляра класса **TOperation**, в модуле **Unit_operation** нами не определялся. Последнее означает, что данный метод унаследован от базового класса **TObject**. Таким образом, выполнение кода "**TOperation.Create**" означает обращение из класса-потомка **TOperation** к унаследованному методу **Create**.

Перегрузка виртуального метода.

В предыдущем параграфе рассматривалась перегрузка статических методов. Представляет интерес проследить, что следует изменить для перегрузки метода виртуального.

В модуль **Unit_operation** добавим определение класса-потомка **TOper_child**. В классе-потомке определим метод **Summ**, которым обладает класс-родитель. Одноименный метод в обоих классах перегрузим.

В результате проведённых преобразований модуль **Unit_operation** примет вид:

Листинг 6.3.

```
Unit Unit_operation;
interface
 type
  TOperation = class(TObject)
   public
    Function summ(a,b:integer): integer; overload; virtual;
  end;

  TOper_child = class(TOperation)
   Function diff(a,b:integer):integer;
   Function summ(a,b,c: integer): integer; overload;
  end;

implementation

Function TOperation.summ(a,b: integer): integer;
  Begin  summ:= a+b;  end;

Function TOper_child.diff(a,b:integer):integer;
  Begin  diff:= a-b;  end;

Function TOper_child.summ(a,b,c:integer):integer;
  begin  summ:= a+b+c;  end;
end.
```

Сопроводим приведённый код кратким комментарием:

– метод **summ** класса-родителя в строке заголовке помечен двумя ключевыми словами "**overload**" и "**virtual**": "**Function summ(a,b:integer): integer; overload; virtual;**";

– метод **summ** класса-потомка в строке заголовке помечен одним ключевым словом "**overload**": "**Function summ(a,b,c: integer): integer; overload;**". Следует обратить внимание на то, что класс-потомок наследует виртуальность метода **summ** класса-родителя, поэтому данный метод в классе-потомке ключевым словом "**virtual**" не помечается.

Изменения модуля **Unit_Program** отражены в листинге 6.4:

Листинг 6.4.

```
Unit Unit_Program;
interface
. . .
implementation
{$R *.dfm}
uses Unit_operation;
var  Sm: TOperation;
    Df: TOper_child;

procedure TForm_Program.Panel1Click(Sender: TObject);
var x,y,z,r,q: Integer;
begin
  x:= StrToInt(Edit1.Text);
  y:= StrToInt(Edit2.Text);
  z:= StrToInt(Edit3.Text);

  Df:= TOper_child.Create;

  r:= Df.Summ(x,y);
  q:= Df.Summ(x,y,z);
  Label1.Caption:= IntToStr(r);
  Label2.Caption:= IntToStr(q);
end;
end.
```

Комментарий к приведённому выше коду:

– ключевая строка кода, обеспечивающая создание экземпляра класса **TOper_child**: " **Df:= TOper_child.Create;**";

– созданный экземпляр класса **TOper_child** сохраняется в переменной объектного типа **Df**;

– следует обратить внимание на то, что для создания экземпляра класса-потомка совсем не требуется предварительно создавать экземпляр класса-родителя;

– метод **Create**, обеспечивающий создание экземпляра класса **TOper_child**, в модуле **Unit_operation** не определялся. Это означает, что данный метод унаследован от родителя класса-родителя, то есть от класса **TObject**. Таким образом, выполнение кода "**TOper_child.Create**" означает обращение из класса-потомка 2-го колена **TOper_child** к унаследованному методу **Create**.

## Задания для самостоятельной работы

1. Создать класс **TOper_mult**, в котором определить динамический (**dynamic**) метод **mult**, возвращающий произведение подаваемых на его вход 2-х аргументов.

2. Создать класс-потомок **TMult_mult** класса **TOper_mult**, в котором доопределить ещё один перегружаемый метод **Mult** с 3 параметрами.

## Определение полей, методов и свойств класса в Lazarus.

В первой части пособия рассмотрение велось на примерах классов простейшего типа, обладающих только методами. Эти классы не имели полей. Последнее означает, что экземпляры этих классов были лишены возможности хранить какие-либо данные.

Начиная с данного параграфа конструируемые классы наделяются полями и свойствами.

Как отмечалось в первой части пособия, важнейшее свойство объектно-ориентированного программирования – инкапсуляция (управление доступом к информации, объединённой под эгидой какого-либо класса).

Информация, благодаря которой класс является тем, чем он является, хранится в полях класса. Поля класса – контейнеры, в которых хранится информация. Доступ к этим контейнерам строго регламентируется. Непосредственные манипуляции с содержимым контейнеров не предусмотрены.

Для доступа к полям класса применяются инструменты двух типов: методы и свойства.

Методы позволяют выполнять два типа действий:

- копировать данные, хранящиеся в полях (при этом содержимое полей не меняется);
- изменять содержимое полей (грамотно сконструированный метод, как правило, проверяет: является ли новое значение безопасным для соответствующего поля).

Свойства являются представителями полей с четко очерченными полномочиями. Свойства не обладают собственными ресурсами, позволяющими хранить данные. Данные могут храниться только в полях. Свойства представляют собою ссылки на соответствующие поля данных.

Свойства позволяют выполнять два типа операций:

- ″считывать″ (ключевое слово **read**) данные, хранящихся в полях, с целью каких-либо последующих манипуляций с ними (либо непосредственно без метода-посредника, либо через посредника, в роли которого выступает соответствующий метод). При этом свойство – всего лишь ссылка на соответствующее поле. В операции принимает участие значение поля, но не значение свойства;
- ″присваивать″ передаваемые значения (ключевое слово **write**) соответствующим полям, изменяя при этом их содержимое (либо непосредственно без метода-посредника, либо пользуясь услугами соответствующего метода-посредника). При этом свойство – ссылка, указывающая, какому полю должно быть присвоено соответствующее значение.

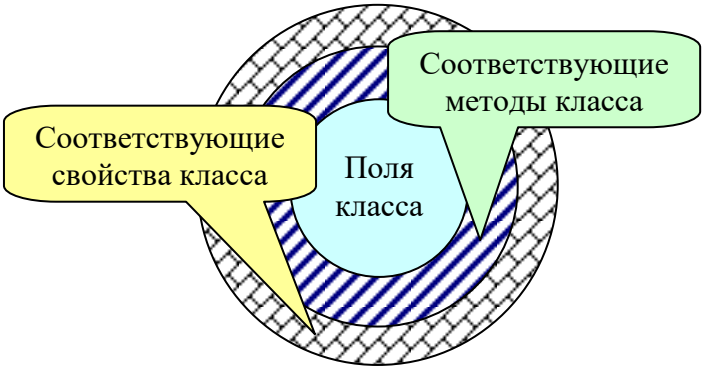



Рис. 1. Соотношение полей, методов и свойств класса.

Приведённая выше схема иллюстрирует следующее:

- методы класса – посредник между полями и свойствами;
- свойства, в свою очередь – своеобразный интерфейс, выстроенный над полями и методами.

Рассмотрение перечисленных выше особенностей полей и свойств начнём с примера построения класса **TSumma**, предназначенного для решения следующей задачи:

| определить значение суммы элементов целочисленного массива. |
|---|

Входные параметры: количество элементов массива и массив значений этих элементов.

Выходной параметр: сумма значений элементов целочисленного массива.

Построим класс **TSumma** (потомок базового класса **TObject**), который бы позволил решить проставленную задачу. Определение класса **TSumma** разместим в модуле **Unit_summa_row**.

Листинг 8.1.

```
unit Unit_summa_row;
interface
 uses SysUtils;
 const max = 10;
 type
  TList = array[1..max] of string;
  TSumma = class(TObject)
   Private
    FL: integer;
    FO: TList;
    Function GetO(i: Integer): string;
    Procedure SetO(i: Integer; v: string);
   Public
    property L: Integer read FL write FL;
    property O[i: Integer]: string read GetO write SetO;
    Function summ(L: integer): integer;
    Constructor Init_summ;
  end;
implementation
Constructor TSumma.Init_summ;
 Begin  L:= 1;  end;
Function TSumma.GetO(i: Integer): string;
 Begin  Result:= FO[i];  end;
Procedure TSumma.SetO(i: Integer; v: string);
 Begin  FO[i]:= v;  end;
Function TSumma.summ(L: integer): integer;
var i, S: integer;
 begin
  S:= 0;
```

```
  For i:= 1 to L Do S:= S + StrToInt(O[i]);
  Result:= S
 end;
end.
```

Прокомментируем приведённый выше код:

– наименование модуля - "**Unit_summa_row**", он сохраняется в файле с именем "**Unit_summa_row.pas**";

– в разделе **interface** модуля размещены секции **uses**, **const** и **type**;

– в секции **uses** нашего модуля к ресурсам системного модуля **System**, подключаемого при выполнении компиляции по умолчанию, подключаются ресурсы ещё одного системного модуля – **SysUtils**. В этом модуле находится определение функции **StrToInt**, которая используется при расшифровке метода **summ** класса **TSumma**;

– в секции **const** нашего модуля определяется константа **max (= 10**), задающая предельное количество суммируемых элементов массива;

– в секции **type** нашего модуля приводится определение одномерного строкового массива как типа **TList**, являющегося внешним по отношению к полям, свойствам и методам класса **TSumma**. В этом определении используется определённая ранее константа **max= 10**;

– центральное определение секции **type** нашего модуля – определение класса **TSumma**. Данное определение представлено двумя разделами: **Private** (закрытый) и **Public** (общедоступный);

– в разделе **Private** определены два поля класса **TSumma**: **FL** и **FO**, а также два метода **GetO** и **SetO**, предназначенные для работы с полем **FO**;

– в Lazarus, как правило, определения полей предшествуют определениям методов и свойств. Имена полей классов в Lazarus принято начинать буквой **F** (от **Field** - поле). Это удобно, позволяя отличать наименования полей от наименований свойств и методов;

– поле **FL** предназначено для хранения количества элементов массива целочисленных значений. С этим полем связано свойство **L** , поддерживающее манипуляции со значениями поля **FL**. Определение свойства **L** приведено в разделе **Public**:

**property L: Integer read FL write FL;**

– значение **FL** за ключевыми словами **read** и **write** показывает, что свойство **L** замкнуто непосредственно на поле **FL** (минуя методы-посредники), то есть **L** не более чем представитель поля **FL**;

– поле **FO** предназначено для хранения массива целочисленных значений, которые требуется просуммировать. Доступ к элементам массива обеспечивается методами **GetO(index)** и **SetO(index, value)**, определёнными в том же разделе;

– метод **GetO(index)** реализован в виде функции, возвращающей значение элемента с номером **index** массива поля **FO**:

**Function GetO(i: Integer): string;**

– метод **SetO(index, value)** реализован в виде процедуры, которая обеспечивает присваивание значения **value** элементу с номером **index** массива поля **FO**:

**Procedure SetO(i:Integer; v:string);**

– методы **GetO** и **SetO**, определённые в разделе **Private** и предназначенные для работы с полем **FO**, замкнуты на свойство **O[index]**, определённое в разделе **Public**:

**property O[i:Integer]:string**
**read GetO write SetO;**

– свойство **O[index]** – единственный канал влияния (раздел **Public** !) на поле **FO**, представляющее собою массив подлежащих суммированию чисел. Это свойство не имеет прямого выхода на поле **FO**. Доступ к элементам массива этого поля – только через

методы **GetO** и **SetO**. Правомерен вопрос: нельзя ли организовать прямой интерфейс, связав поле **FO** со свойством **O** напрямую, без посредников. Ответ отрицательный. На массивы, используемые при определении классов, налагается ряд ограничений. Одно из ограничений касается массива свойств класса: спецификации доступа **read** и **write** не могут ссылаться на поле, представляющее собою массив. Они обязательно должны ссылаться на методы **Get** и **Set**, предоставляющие корректный доступ к элементам этого массива;

– метод **summ** обеспечивает суммирование значений элементов массива. Доступ к этим элементам обеспечивается свойством **O[index]**. Рассчитанное значение возвращается (ключевое слово **Result**) функцией **summ**:

```
S:= 0;
For i:= 1 to L Do S:= S+StrToInt(O[i]);
Result:= S
```

– последняя процедура **Init_summ** – конструктор класса **TSumma**, обеспечивающий создание экземпляра этого класса и инициализацию массива элементов. Создаётся массив, содержащий единственный элемент, равный **0**.

Использование сконструированного класса.

Для использования возможностей организованного нами класса **TSumma** необходимо создать соответствующий **Lazarus**-проект. В состав этого проекта должны войти файлы:

– файл, в котором располагается определение класса **TSumma**. Имя этого файла - "**Unit_summa_row.pas**";

– файл, в котором располагаются средства управления возможностями класса **TSumma**. Имя этого файла - "**Unit_Program.pas**". Рассмотрим его структуру:

Листинг 8.2.

```
unit Unit_Program;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, ExtCtrls, Unit_summa_row,;
type
  TForm1 = class(TForm)
    Edit2: TEdit;
    Panel1: TPanel;
    Label1: TLabel;
    Panel2: TPanel;
    Panel3: TPanel;
    Label2: TLabel;
    Edit1: TEdit;
    Panel4: TPanel;
    procedure FormCreate(Sender: TObject);
    procedure Edit1Exit(Sender: TObject);
    procedure Panel1Click(Sender: TObject);
    procedure Panel2Click(Sender: TObject);
    procedure Panel3Click(Sender: TObject);
    procedure Panel4Click(Sender: TObject);
  end;
var
  Form1: TForm1;
  Sm: TSumma;
  n: Integer;
implementation
```

```
{$R *.dfm}
procedure TForm1.FormCreate(Sender:TObject );
begin
  n:= 1;
  Label2.Caption:= IntToStr(n);
end;
procedure TForm1.Panel1Click(Sender:TObject );
var r: LongInt;
begin
  r:= Sm.Summ(Sm.L);
  Label1.Caption:= IntToStr(r);
end;
procedure TForm1.Panel2Click(Sender:TObject );
begin
  Sm.O[n]:= Edit2.Text;
  If n>1 then
   begin
     n:= n-1;
     Label2.Caption:= IntToStr(n);
     Edit2.Text:= Sm.O[n];
   end;
end;
procedure TForm1.Panel3Click(Sender:TObject );
begin
  Sm.O[n]:= Edit2.Text;
  If n < Sm.L then
   begin
     n:= n+1;
     Label2.Caption:= IntToStr(n);
     Edit2.Text:= Sm.O[n];
   end;
end;
procedure TForm1.Edit1Exit(Sender: TObject);
var i: integer;
begin
  Sm.L:= StrToInt(Edit1.Text);
  For i:=1 to Sm.L Do Sm.O[i]:=IntToStr(0);
  Edit1.Enabled:= false;
end;
procedure TForm1.Panel4Click(Sender: TObject );
begin
  Sm:= TSumma.Create;
  ShowMessage('Create: Ok');
end;
end.
```

Комментарий к приведённому выше коду:

– в разделе **interface** модуля располагаются секции **uses**, **type** и **var**;

– к перечню модулей в секции **uses** следует добавить модуль **Unit_summa_row**, в котором определён организованный нами класс **TSumma**;

– в секции **type** дано определение класса **TForm1**. На форме расположены два текстовых окна **TEdit** (для ввода количества элементов массива (не более **max = 10**) и ввода самих элементов массива). Индекс элемента массива выводится на метку **TLabel**. Изменение значения индекса элемента массива осуществляется кодом, связанным с панелями **TPanel**. Метка **TLabel** предназначена для публикации результата суммирования (метод **Summ**);

– в секции **var** определена переменная **Sm** класса **TSumma**, предназначенная для хранения экземпляров этого класса;

– в секции **var** определена и переменная **n**, служащая для хранения текущего значения индекса элемента массива;

– в разделе **implementation** модуля располагаются расшифровки методов, объявленных в секции **type** раздела **interface**;

– создание экземпляра класса осуществляется процедурой **Panel4Click**:

```
Sm:= TSumma.Create;
```

– уменьшение индекса текущего элемента массива осуществляется процедурой **Panel2Click**:

```
Sm.O[n]:= Edit2.Text;
If n>1 then
begin
 n:=n-1;Label2.Caption:= IntToStr(n);
 Edit2.Text:= Sm.O[n];
end;
```

– увеличение индекса текущего элемента массива осуществляется процедурой **Panel3Click**:

```
Sm.O[n]:= Edit2.Text;
If n < Sm.L then
begin
 n:= n+1; Label2.Caption:= IntToStr(n);
 Edit2.Text:= Sm.O[n];
end;
```

– код процедуры **Edit1Exit** обеспечивает задание количества используемых элементов массива и инициализацию этих элементов:

```
Sm.L:= StrToInt(Edit1.Text);
For i:=1 to Sm.L Do Sm.O[i]:=IntToStr(0);
```

– код процедуры **Panel1Click** обеспечивает расчёт значения искомой суммы:

```
r:= Sm.Summ(Sm.L);
Label1.Caption:= IntToStr(r);
```

Запуск подготовленного проекта.

Запуск откомпилированного проекта на исполнение (результат компиляции находится в файле "**Class_definition.exe**") приводит к следующему результату. Перед пользователем раскрывается окно программы, в котором представленные кнопки предназначены для выполнения следующих действий:

**Start** – запуск процедуры создания экземпляра класса **TSumma** (**Panel4Click**);

< и > – запуск процедур уменьшения и увеличения индекса текущего элемента массива (осуществляется процедурами **Panel2Click** и **Panel3Click** соответственно**);**

**Calc** – запуск процедуры расчёта значения искомой суммы (**Panel1Click**).

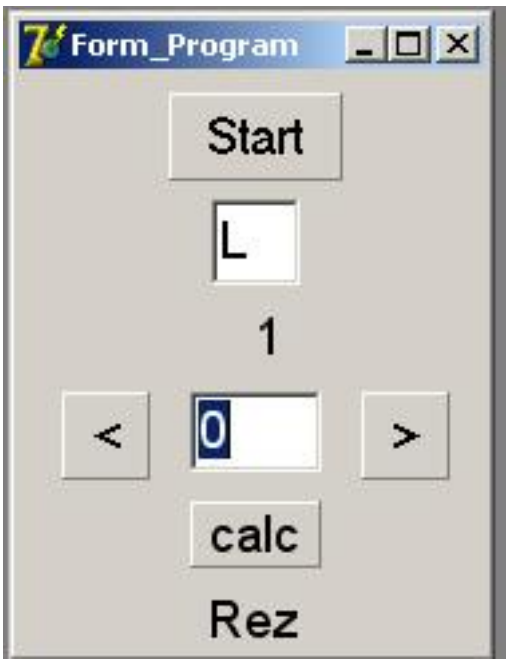



Рис. 2. Вид формы проекта после запуска на исполнение.

Результат щелчка по кнопке **Start** представлен на приводимом ниже рисунке.


Рис. 3. Появление сообщения о создании экземпляра класса **TSumma**.

После создания экземпляра класса следует определиться с количеством элементов массива. Для этого в окно **L** вводится соответствующее значение. В данном случае – 3.

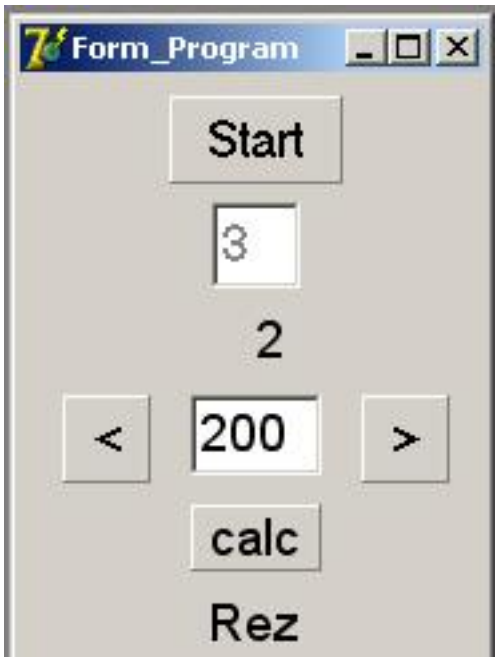



Рис. 4. Вид формы проекта после ввода количества элементов массива.

В окно просмотра элементов массива выводятся значения элементов массива. В этом же окне выведенные значения можно изменять.

Результат суммирования выводится щелчком по кнопке **Calc**.

## Задания для самостоятельной работы

1. Модернизировать класс TOperation, добавив ещё один метод (Diff(a,b)), который бы возвращал разность поданных на его вход аргументов. В модуль Unit_Program внести дополнения, позволяющие использовать метод Diff.
2. Создать новый класс TOper_mult, в котором определить метод (Mult(a,b,c)), возвращающий произведение поданных на его вход аргументов. Создать модуль, позволяющий использовать метод Mult.
3. Создать новый проект, перенеся определение класса TOper_mult в модуль формы. Обеспечить возможность использования метода Mult.

## Темы кейсов (разработка классов, объектов, методов и свойств):

Тема 1: Создание приложения «Сапёр»

Тема 2: Программный комплекс для решения систем линейных алгебраических уравнений (3 уравнения)

Тема 3: Программный комплекс для решения обыкновенных дифференциальных уравнений

Тема 4: Игра «Судоку»

Тема 5: Операции с матрицами

Тема 6: Игра «Пятнашки»

Тема 7: Моделирование шифратора и дешифратора для 16-й системы счисления на основе логических элементов.

Тема 8: Операции с дробями

Тема 9: Игра «Морской бой»

Тема 10: Игра «Крестики-Нолики»