Введение в Delphi

1. История версий
2. Основы ООП
3. Структура класса

История версий

- **Borland Delphi 1** была предназначена для разработки под 16-разрядную платформу Win16;
- Начиная с **Borland Delphi 2** компилируются программы под 32-разрядную платформу Win32;
- Вместе с **Borland Delphi 6** выходит совместимая с ним по языку и библиотекам среда **Kylix**, предназначенная для компиляции программ под операционную систему Linux;
- С 2006 года **Borland** передает подразделения, занимающиеся средствами разработки, своей дочерней компании **CodeGear**, которая в 2008 году была продана компании **Embarcadero Technologies**

История версий

Delphi for .NET — среда разработки Delphi, а также язык Delphi (Object Pascal), ориентированные на разработку приложений для .NET.

Первая версия полноценной среды разработки Delphi для .NET — **Delphi 8**. Она позволяла писать приложения только для .NET.

В **Delphi 2006**, можно писать приложения для .NET, используя стандартную библиотеку классов .NET, VCL для .NET. Delphi 2006 содержит функции для написания обычных приложений с использованием библиотек VCL и CLX.

Начиная с версии 2009, поддержка Delphi.NET была прекращена. Для разработки под .NET предлагается **Delphi Prism**.

История версий

В 2008 году Embarcadero публикует пресс-релиз на **Delphi for Win32 2009**.

Нововведения:

- По умолчанию полная поддержка Юникода во всех частях языка, VCL и RTL;
- Обобщённые типы.
- Анонимные методы.
- Функция Exit теперь может принимать параметры в соответствии с типом функции.

История версий

- В 2009 году выходит интегрированная среда разработки **Embarcadero Rad Studio 2010**, в которую вошла новая версия **Delphi 2010**.
- Новое в Delphi 2010
  - Delphi 2010 включает свыше 120 усовершенствований для повышения производительности.
  - IDE Insight в Delphi 2010 — мгновенный доступ к любой функции или параметру.
  - Классический интерфейс Delphi 7 и панель инструментов со вкладками как опция.
  - Расширение RTTI - поддержка атрибутов, которые могут быть применены к типам(в том числе классам и интерфейсам), полям, свойствам, методам и к элементам перечислений

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Объектно-ориентированное программирование (ООП)

– это методология программирования, основанная на представлении программы в виде совокупности ***объектов***, каждый из которых является экземпляром определенного ***класса***, а классы образуют иерархию ***наследования***.

Объектно-ориентированное программирование (ООП)

Основное преимущество ООП является возможность многократного использования программного кода.

Объектно-ориентированное программирование

Объект – это опознаваемый предмет, блок или сущность, имеющие важное функциональное значение в определенной области.

Объектно-ориентированное программирование

Объект обладает: состоянием, поведением и индивидуальностью

- **Состояние** – перечень всех возможных свойств объекты (Property) и их текущими значениями
- **Поведение** характеризует то, как один объект воздействует на другие или как он подвергается их воздействию в постоянно меняющихся условиях
- **Индивидуальность** – свойство или набор свойств, благодаря которым различаются между собой объекты одного класса.

Классы

Классы – это специальные типы, которые содержат поля, методы и свойства.

Класс объявляется с помощью зарезервированного слова **class**.

```
Type
    TMyClass = class
        {поля}…
        {методы}…
        {свойства}…
    end;
```

Классы

- Важным отличием классов от других типов данных является то, что объекты класса всегда располагаются в динамической памяти (куче), поэтому объект-переменная представляет собой лишь указатель на динамическую область памяти. Однако при ссылке на содержимое объекта запрещается использовать символ «^» за именем объекта.

Классы

Класс служит образцом для создания конкретных экземпляров – **объектов**

Любой объект является экземпляром класса.

Объявление экземпляра класса:

```
var
  myClass: TMyClass;
```

Классы

- Интерфейс класса соответствует его внешнему проявлению и подчеркивает его абстрактность.
- Реализация класса составляет его внутреннее проявление и определяет особенность его поведения.

Объектно-ориентированное программирование

В основе объектно-ориентированного программирования лежат три фундаментальных принципа:

- **инкапсуляция**
- **наследование**
- **полиморфизм**.

Инкапсуляция

Инкапсуляция

- Класс представляет собой единство трех сущностей – **полей**, **методов** и **свойств**.
- Объединение этих сущностей в единое целое и называется **инкапсуляцией**.
- Инкапсуляция позволяет во многом изолировать класс от остальных частей программы, сделать его «самодостаточным» для решения конкретной задачи.

Инкапсуляция

- Инкапсуляция полезна потому, что помогает перечислить связанные методы и данные под одной эгидой, а так же скрывать детали реализации, которые не требуют своей демонстрации или могут быть изменены в будущих версиях объекта.

Инкапсуляция

- Хорошо сконструированные объекты должны состоять из двух частей:

  1) Данных и разделов реализации, скрытых от программистов, использующих объект (с целью защиты данных от несанкционированных изменений)

  2) Набора интерфейсных элементов, предоставляющих возможность программистам обращаться со скрытыми методами и данными.

Наследование

- Любой класс может быть порожден от другого класса. Для этого при его объявлении указывается имя класса родителя:

TChildClass = class (TParentClass)

Наследование

- Дочерний класс автоматически наследует поля, методы и свойства своего родителя и может добавлять их новыми.
- Таким образом, принцип наследования обеспечивает поэтапное создание сложных классов и разработку собственных библиотек классов.

Наследование

- Все классы Object Pascal порождены от единственного родителя – класса **TObject**.
- Этот класс не имеет полей и свойств, но включает в себя методы самого общего назначения, обеспечивающие весь жизненный цикл любых объектов – от их создания до уничтожения.

Наследование

- Программист не может создать класс, который бы не был дочерним классом TObject, т.к. следующие два объявления идентичны:

TMyClass= class (TObject)

TMyClass= class

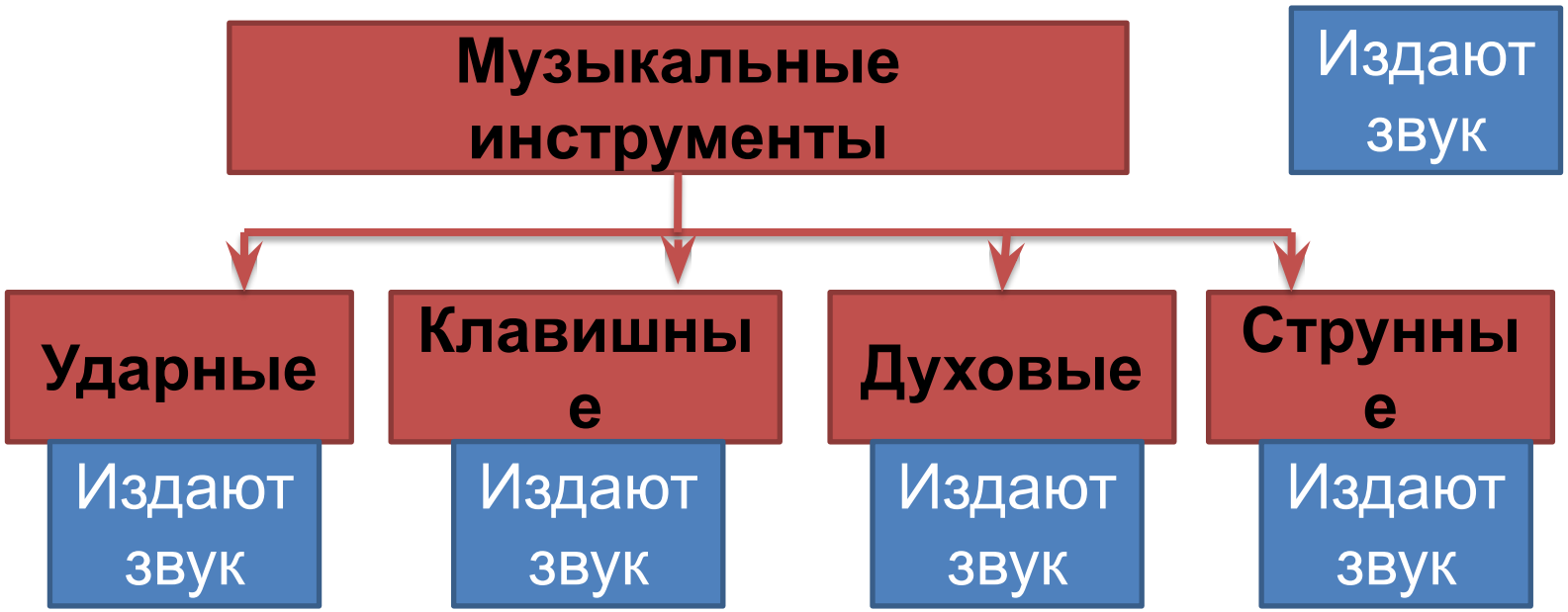
Наследование

- Принцип наследования приводит к созданию ветвящегося дерева классов.
- Каждый потомок дополняет возможности своих родителей и передает их своим потомкам.

Полиморфизм

- Полиморфизм – это свойство классов решать схожие по смыслу проблемы различными способами.
- В рамках Delphi поведение класса определяется набором входящих в него методов.
- Изменяя алгоритм того или иного метода в потомках класса, можно придавать этим потомкам отсутствующие у родителя специфические свойства.

Полимоорфизм



Полиморфизм

- Для изменения метода необходимо **перекрыть** его в потомке, т.е. объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте- родителе и объекте-потомке будут действовать два одноименных метода, имеющих различные алгоритмы.
- Полиморфизм достигается также **виртуализацией** методов, позволяющей родительским методам обращаться к методам своих потомков.

СТРУКТУРА ОПИСАНИЯ КЛАССА

Структура описания класса

```
Type
    TMyClass = class
        {поля}…
        {методы}…
        {свойства}…
    end;
```

Поля

- Полями называются инкапсулированные в класса данные. Поля могут быть любого типа, в том числе классами. Принято имя поля начинать с буквы F (Field). Например:

```
type
  TMyClass = class(TForm)
  FInt:integer;
  fWidth:word;
  FPrim1:TObject;
  fFam:string;
  end;
```

Поля

- Каждый объект получает уникальный набор значений полей, но общий для всех объектов для данного класса набор методов и свойств.
- Понятие инкапсуляции и хороший стиль объектно-ориентированного программирования требуют, чтобы обращение к полям объектов выполнялось исключительно посредством методов.
- Однако в Delphi разрешается обращаться к полям и напрямую. Для этого используются составные имена полей:

```
Var MyObj1: TMyClass;


…
begin
…
   MyObj1.ffam:=’Иванов’;
   MyObj1.fwidth:=500;
…
end;
```

Поля

Класс-потомок получает все поля всех своих предков и может пополнить их своими, но он не может переопределить их или удалить.

Методы

- Инкапсулированные в классе процедуры и функции называются методами. Они объявляются, так же как и обычные подпрограммы:

```
type
  TForm1 = class(TForm)
     procedure FormClick(Sender: TObject);
     function KeyDown : Word;
  end;
```

Методы

- Для реализации методов нужно описать соответствующую подпрограмму в разделе реализации модуля. При этом в заголовке метода указывается название класса.

Методы

```
Implemenation
  …
procedure TForm1.FormClick(Sender:TObject);
begin
   {реализация метода FormClick}
end;

function  TForm1.KeyDown:Word;
begin
   {реализация метода KeyDown}
end;
….
end.
```

Методы

Доступ к методам класса осуществляется с помощью составных имен:

```
Var  Form1: TForm1;

Begin
…
Form1.FormClick(…)
Key:= Form1.KeyDown;
…
End;
```

Свойства

- Свойства – это специальный механизм классов, регулирующий доступ к полям.
- При работе с объектом свойства выглядят как поля: они принимают значения и участвуют в выражениях.
- Но в отличии от полей свойства не занимают место в памяти, а операции их чтения и записи ассоциируются с обычными полями и методами.
- Это позволяет создавать необходимые сопутствующие эффекты при обращении к свойствам.

Свойства

- Объявление свойства выполняется с помощью зарезервированных слов **property**, **read**, **write**.

- Обычно свойство связано с некоторым полем и указывает те методы класса, которые должны использоваться при записи в это поле или при чтении из него.

Свойства

- Например,

```
Type
   TPeople = class
        FName: string;
        procedure GetName;
        property Name: string read FName write GetName;
   end;
```

Свойства

Обращение к свойствам выглядит в программе как обращение к полям:

```
var People: TPeople;
   Get: string;
   ...
   People.Name := 'Сергей';
   Get := People.Name;
```

Свойства

Если один из спецификаторов доступа опущен, то значение свойства можно либо только читать (задан спецификатор read), либо только записывать (задан спецификатор write).

```
type
   TPeople = class
     FName: array of string;
     function GetName: integer;
     property Name: integer read GetName;
   end;


function TPeople.GetName: integer;
   begin
      Result := Length(FName);
   end;
```

Свойства

- В отличии от полей свойства не имеют адреса в памяти, поэтому к ним запрещено применять операцию @.
- Как следствие, их нельзя передавать в var- и out-параметрах процедур и функций.
- Технология объектно-ориентированного программирования в среде Delphi предписывает избегать прямого обращения к полям, создавая вместо этого соответствующие свойства.

Свойства

- Методы получения (чтения) и установки (записи) значений свойств подчиняются определенным правилам.
- Метод **чтения** свойства - это всегда **функция**, возвращающая значение того же типа, что и тип свойства.
- Метод **записи** свойства - это обязательно **процедура**, принимающая параметр того же типа, что и тип свойства.
- В остальных отношениях это обычные методы объекта.

Свойства

```
type
  TPeople = class
    FName: boolean;

    procedure SetName(const AName: boolean);
    function GetName: integer;

    property Name: boolean read FName write SetName;
    property Count: integer read GetName;
  end;
```

Свойства

- Использование методов для получения и установки свойств позволяет проверить корректность значения свойства, сделать дополнительные вычисления, установить значения зависимых полей и так далее.

```
procedure TPeople.SetName(const AName: boolean);
begin
        if Name <> AName then
          begin
           if FName then // Если состояние
                                  изменяется то
              ...
           else
              ...
            FName := AName; // Сохранение
                                  состояния в поле

          end;
end;
```

Основные секции класса

- В интерфейсе класса выделяются отдельные секции, определяющие область видимости элементов класса.
- Внутри каждой секции вначале определяются поля, а затем методы и свойства.

Основные секции класса

- **Public** (общедоступная) – ее могут использовать все пользователи объектов данного класса;
- **Private** (личная) – может использоваться внутри реализации данного класса;
- **Protected** (защищенная) – доступна только классам, которые являются потомками данного класса, а также методам самого класса;

Основные секции класса

- **Published** (опубликованная) – часть класса, аналогичная общедоступной, но имеющая некоторые особенности в реализации. В ней перечисляются свойства, которые доступны не только на этапе исполнения, но и на этапе конструирования программы (т.е. в окне Инспектора Объектов). Она используется только при разработке нестандартных компонентов

```
Unit Unit1;
Interface
Uses  Controls, Forms;
Type
      TForm1=class(TForm)
         Button1:TButton;
  Private
         FintField: Integer;
         Procedure SetValue(Value:
    Integer);
         Function GetValue: Integer;
  Published
       Property IntField: read GetValue
     write SetValue;
  Protected
         Procedure Proc1;
  Public
       Procedure Proc2;
end;
 var
```

```
Implementation
   Procedure TForm1.Proc1;
   begin
      button1.color := clBtnFace;
      FintField := 0;
      IntField := 0;
      Proc1;
      Proc2;
   end;

begin
Form1.button1.color :=
    clBtnFace;
Form1.FintField := 0
Form1.IntField := 0;
Form1.Proc1;
Form1.Proc2;
end.
```

//ошибка

```
Unit unit2;
Interface
Uses controls, unit1;
Type
  TForm2=class(TForm1);
    Button2:Tbutton;
   Procedure Proc3(Sender:Tobject);
       end;
var
       Form2: Tform2;
```

```
Implementation
Procedure
     TForm2.Proc3(Sender:TObject);
begin
       Button2.Color := clBTNFace;
       FintField := 0;
       IntField := 0;
       Proc1;
       Proc2;     //ошибка
end;

begin
  Form1.button1.color := clBtnFace;
Form1.FintField := 0;
Form1.IntField := 0;
Form1.proc1;
Form1.proc2;     //ошибка
end.
```

КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

Конструкторы и деструкторы

- В состав любого класса входят два специальных метода – конструктор и деструктор.
- Конструктор распределят объект в динамической памяти и помещает адрес этой памяти в переменную *Self*, которая автоматически объявляется в классе.
- Деструктор удаляет объект из кучи.

Конструкторы и деструкторы

- У класса T*Object* эти методы называются *Create* и *Destroy*, так же они называются в подавляющем большинстве его потомков.
- По своей форме конструкторы и деструкторы являются процедурами, но объявляются с помощью зарезервированных слов **Constructor** и **Destructor**

```
Type
   TMyClass = class
   IntField: Integer;
   Constructor Create (Value: Integer);
    Destructor Destroy;
End;
 …
Constructor Create(Value: Integer);
   begin
        IntField := Value;
   end;

Destructor Destroy;
   begin
     …
   end;
```

Конструкторы и деструкторы

- Обращение к конструктору должно предварять любое обращение к полям и некоторым методам объекта.
- В базовом классе *TObject* определен метод *Free*, который сначала проверяет действительность адреса объекта и лишь затем вызывает деструктор *Destroy*.
- Обращение к деструктору объекта будет ошибочным, если объект не создан конструктором, поэтому для уничтожения ненужного объекта следует вызывать метод *Free*

```
Var
   MyObj: TMyClass;
Begin
 MyObj.IntField := 0;

 MyObj := TMyClass.Create;

 MyObj.IntField := 0;

 . . . . .
   MyObj.Free;

End;
```

Ошибка! *Объект не создан конструктором*

Вызов конструктора. *Создаем объект в памяти*

Правильное обращение к полю

Уничтожаем ненужный объект

Конструкторы и деструкторы

- Конструктор будет создавать объект, только если при вызове перед его именем указывается имя класса.
- Если же указывается имя существующего объекта, то создание объекта не происходит, а лишь выполняется код, содержащийся в теле конструктора

```
 …
Constructor Create(Value: Integer);
  begin
     IntField := Value;
     If Value mod 3 = 0 then
         IntField := IntField div 3;
  end;
 …
Var
    Obj1, Obj2:TMyClass;
begin
…
   Obj1.Create(5);

   Obj2 := TMyClass.Create(4);

   Obj2.Create(6);
end.
```

Ошибка! *Объект Obj1 не существует*

Создаем Obj2

Выполняем действия конструктора

Конструкторы и деструкторы

- Если при объявлении класса конструктор / деструктор не описаны, то для создания/удаления будут вызываться соответствующие методы родительского класса.
- Т.к. все классы наследуются от TObject, то любой класс имеет конструктор/деструктор по умолчанию
- Создавайте свои конструкторы/деструкторы, только если требуются дополнительные действия по инициализации или удалению объектов

Конструкторы и деструкторы

- Правила для реализации собственных конструкторов / деструкторов
  - в конструкторе класса-потомка следует сначала вызвать конструктор своего родителя, а уже затем осуществлять дополнительные действия.
  - В последней строке деструктора необходимо вызвать деструктор класса-предка
- Вызов любого метода родительского класса достигается с помощью зарезервированного слова **Inherited**

Возможная реализация конструктора

```
Constructor TMyClass.Create (Value: Integer);

begin
  Inherited Create;
  IntField := Value;
End;
```

Вызываем унаследованный конструктор

Реализуем дополнительные действия

Возможная реализация деструктора

```
Destructor TMyClass.Destroy;

begin
  …
  Inherited Destroy;
End;
```

Реализуем дополнительные действия

Вызываем унаследованный деструктор

Раннее и позднее связывание.

- Методы класса могут перекрываться в потомках.
- Потомок класса может иметь сходную по названию процедуру или функцию, которая будет выполнять другое действие.
- В Object Pascal возможно статическое и динамическое замещение методов.

```
type
 TParent = class
  Fx, Fy:byte;
  constuctor MyConstr;
  procedure WriteFields;
 end;

 TChild = class(TParent)
   Fz: byte;
   constuctor MyConstr;
   procedure WriteFields;
  end;

Constructor
  TParent.MyConstr;
begin
  inherited Create;
  Fx := 0;
  Fy := 0;
end;
```

```
procedure
  TParent.WriteFields;
begin
  Writeln(Fx,' `, Fy);
end;

Constructor
  TChild.MyConstr;
begin
  inherited MyConstr;
  Fz := 100;
end;


procedure
  TChild.WriteFields;
begin
  Writeln(Fx, ` ', Fy,
  ` ', Fz);
end;
```

```
var
    _par: TParent;
    _child: TChild;

begin
 _par:= TParent.MyConstr;

 _child:=TChild.MyConstr;

 _par.WriteFields;

 _child.WriteFields;

end.
```

Объявляем экземпляры классов TParent и TChild

Вызов родительского метода: WriteFields

Результат: 0 0

Вызов дочернего метода: WriteFields

Результат: 0 0 100

- Важно понимать, что на этапе выполнения программы объект представляет собой единое целое, не разделенное на части предка и потомка.

- Во время выполнения программы объекты хранятся в отдельных переменных, массивах и др.
- Во многих случаях удобно *оперировать объектами одной иерархии единообразно*, то есть использовать один и тот же программный код для работы с экземплярами разных классов.

- В Object Pascal объекту родительского класса можно присвоить любой дочерний объект.
- При этом все поля, свойства и методы родительского объекта будут заполнены правильно.
- Обратное утверждение НЕВЕРНО.

_par:=_child;

- При этом присваивании все поля, методы и свойства, которые не входят в TParent игнорируются

- Объект класса TObject совместим с любым другим объектом Delphi.

```
Пример:
var obj:TObject;
    my_obj : TChild;
…
obj := my_obj;
…
my_obj := obj;
```

Допустимое присваивание

Ошибка! Недопустимое присваивание

- возможность доступа к элементам класса определяется **типом ссылки**, а не типом объекта, на который он указывает.

```
var
    _par, _par1: TParent;
    _child: TChild;

begin
  _par:= TParent.MyConstr;
  _child:= TChild.MyConstr;

  _par.WriteFields;
  _child.WriteFields;


  _par := _child;
  _par1 := TChild.MyConstr;

  _par.WriteFields;
  _par1.WriteFields;
end.
```

Вызов родительского метода: WriteFields

Результат: 0 0

Вызов дочернего метода: WriteFields

Результат: 0 0 100

Вызов родительского метода: WriteFields

Результат: 0 0

Результат: 0 0

- Компилятор должен еще до выполнения программы решить какой метод вызывать, и вставить в код фрагмент, передающий управление на этот метод.
- Этот процесс называется **ранним связыванием** (статическое замещение методов) .
- При этом компилятор может руководствоваться только типом переменной, для которой вызывается метод или свойство.
- То, что в этой переменной в разные моменты времени могут находиться ссылки на объекты разных типов, компилятор учесть не может.

- Чтобы вызываемые методы соответствовали типу объекта, необходимо отложить процесс связывания до этапа выполнения программы, а точнее до момента вызова метода, когда уже точно известно, на объект какого типа указывает ссылка.
- Такой механизм называется **поздним связыванием** (динамическое замещение методов) и реализуется с помощью так называемых виртуальных (или динамических) методов.

- Для реализации динамического замещения методов, метод замещаемый в родительском классе, должен объявляться как динамический (с директивой **dynamic**) или виртуальный (с директивой **virtual**).
- Для реализации позднего связывания необходимо, чтобы адреса виртуальных и динамических методов хранились там, где ими можно будет в любой момент воспользоваться.
- Поэтому компилятор формирует специальные таблицы: таблицу виртуальных методов (**Virtual Method Table, VMT**) и таблицу динамических методов (**Dynamic Method Table, DMT**)

- При каждом обращении к замещаемому методу компилятор вставляет код, позволяющий извлечь адрес нужного метода из той или иной таблицы
- В классе-потомке замещающий метод объявляется с директивой **override** (перекрыть)

```
type
 TParent = class
  Fx, Fy:byte;
  constuctor MyConstr;
  procedure WriteFields; virtual;
 end;

 TChild = class(TParent)
   Fz: byte;
   procedure WriteFields; override;
  end;
```

- Получив указание **override**, компилятор создаст код, который на этапе прогона программы поместит в родительскую таблицу точку входа метода класса-потомка, что позволит родителю выполнить нужное действие с помощью нового метода.

```
var
    _par, _par1: TParent;
    _child: TChild;

begin
  _par:= TParent.MyConstr;
  _child:= TChild.MyConstr;

  _par.WriteFields;
  _child.WriteFields;


  _par := _child;
  _par1 := TChild.MyConstr;

  _par.WriteFields;
  _par1.WriteFields;
end.
```

Вызов родительского метода: WriteFields

Результат: 0 0

Вызов дочернего метода: WriteFields

Результат: 0 0 100

Вызов дочернего метода: WriteFields

Результат: ***0 0 100***

Результат: ***0 0 100***

- В результате присваивания **_par:=_child**

при вызове родительского метода WriteFields (**_par.WriteFields**) будет вызываться метод WriteFields дочернего класса.

Однако нельзя вызвать ни один из методов дочернего объекта, который не принадлежит родительскому.

- Разница между динамическими и виртуальными методами состоит в том, что DMT содержит адреса динамических методов только данного класса
- В то время как таблица VMT содержит адреса виртуальных методов не только данного класса, но и всех его родителей
- Значительно большая по размеру таблица VMT обеспечивает более быстрый поиск, в то время как при обращении к динамическому методу программа сначала просматривает таблицу DMT у объекта, затем – у его родителя и т.д., пока не будет найдена нужная реализация метода

```
type
 TParent = class
  Fx, Fy:byte;
  constuctor MyConstr;
  procedure WriteFields; dynamic;
 end;

 TChild = class(TParent)
   Fz: byte;
   procedure WriteFields; override;
  end;
```

Таблица DMT класса TParent
Адрес TParent.WriteFields

Таблица DMT класса TChild
Адрес TChild.WriteFields

```
type
 TParent = class
  Fx, Fy:byte;
  constuctor MyConstr;
  procedure WriteFields; virtual;
 end;

 TChild = class(TParent)
   Fz: byte;
   procedure WriteFields; override;
  end;
```

Таблица VMT класса TParent
Адрес TParent.WriteFields

Таблица VMT класса TChild
Адрес TParent.WriteFields
Адрес TChild.WriteFields

- Вызов динамических и виртуальных методов выполняется через дополнительный этап получения адреса метода из таблиц DMT и VMT, что несколько замедляет выполнение программы.

- **Важно**: Переопределенный метод должен обладать таким же набором параметров, как и одноименный метод базового класса

- Виртуальные (динамические) методы базового класса определяет интерфейс всей иерархии.
- Этот интерфейс может расширяться в потомках за счет добавления новых виртуальных (динамических) методов
- Переопределять виртуальный (динамический) метод в каждом из потомком не обязательно: если он выполняет устраивающие потомка действия, метод наследуется.

- С помощью виртуальных (динамических) методов реализуется **полиморфизм**.

Операции над классами.

- Для классов определены **операции отношения** **=** и **<>**.
- Кроме того, для классов определены еще две операции: **as** (как) и **is** (является).

  Первым операндом в обеих операциях является объект, вторым - класс.

объект as класс

объект is класс

- Если A - объект, а C - класс, то выражение **A as C** возвращает тот же самый объект, но рассматриваемый как объект класса C.
- Операция даст результат, если указанный класс C является классом объекта A или одним из наследников этого класса.
- В противном случае будет сгенерировано исключение.

- Наиболее часто операция **as** применяется к параметру **Sender**, передаваемому во все обработчики событий как объект - источник события и имеющему тип TObject, в котором очень мало свойств для идентификации объекта.

Например:

- **if** (Sender **as** TComponent).Name = 'Button2' **then** ...;

- Выражение **A is C** позволяет определить, относится ли **объект A** к **классу C** или к одному из его потомков. Если относится, то операция **is** возвращает **true**, в противном случае - **false**.
- Например, оператор:

```
if Sender is TButton then ...;
```

- будет реагировать только на объекты класса TButton или потомков этого класса.

Абстрактные методы и классы.

Абстрактные классы

- При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский класс.
- При этом может оказаться, что создавать объекты такого класса не имеет смысла, потому что никакие реальные объекты им не соответствуют.
- Такие классы называются **абстрактными**.

Абстрактные классы

- Абстрактные классы служат только для порождения потомков
- В них задается набор методов, которые каждый из потомков будет реализовывать по-своему.
- Абстрактные методы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах

Абстрактные классы

- Абстрактные классы задают интерфейс для всей иерархии

- При этом методам класса может не соответствовать никаких конкретных действий

- Абстрактные методы объявляются с директивой **abstract**

```
type
 TParent = class
  procedure WriteFields; virtual; abstract;
 end;

 TChild1 = class(TParent)
    FName: string;
    procedure WriteFields; override;
  end;

 TChild2 = class(TParent)
    FAge: byte;
    procedure WriteFields; override;
  end;
```

Абстрактные методы

- Абстрактные методы реализуются только в потомках

- Абстрактные методы должны обязательно перекрываться в потомках

```
//Реализация
 procedure TParent.WriteFields;
 begin
   Write('No fields');
 end;

 procedure TChild1.WriteFields;
 begin
   Write(FName);
 end;

 procedure TChild2.WriteFields;
 begin
   Write(FName);
   Write(Fage);
 end;
```

Ошибка. WriteFields - абстрактный

Ошибка. В классе TChild2 нет поля FName

Методы класса.

Методы класса

- Некоторые методы могут вызываться без создания и инициации объекта.
- Такие методы называются **методами класса**
- Они объявляются с помощью зарезервированного слова **class**

```
Type
  TMyClass = class (TObject)
        Class Function GetClassName: String;
    End;
```

Методы класса

- Доступ к этим методам осуществляется через **имя класса**, а не имя объекта!

```
Type
  TMyClass = class (tObject)
          Class Function GetClassName: String;
      End;
 Var
   S: String;
Begin
   S := TMyClass.GetClassName;
      . . . . . .
End;
```

Методы класса

- **Методы класса** не должны обращаться к полям, т.к. в общем случае вызываются без создания объекта, а следовательно, в момент вызова полей просто не существует.
- Обычно они возвращают служебную информацию о классе – имя класса, имя его родительского класса, адрес метода и т.п.

Одноименные методы

Одноименные методы

- Часто бывает, что методы, реализующие один и тот же алгоритм не должны обращаться к полям, т.к. в общем случае вызываются без создания объекта, а следовательно, в момент вызова полей просто не существует.
- Обычно они возвращают служебную информацию о классе – имя класса, имя его родительского класса, адрес метода и т.п.

Одноименные методы

- В *Delphi* в рамках одного класса можно иметь несколько **одноименных методов**.
- Рассмотренный ранее механизм перекрытия родительского метода одноименным методом потомка приводит к тому, что потомок «не видит» перекрытый родительский метод и может обращаться к нему лишь с помощью зарезервированного слова **Inherited**.

Одноименные методы

- Использование нескольких методов с одним и тем же именем, но с различными наборами параметров называется **перегрузкой методов**.
- Одноименные методы будут доступны если объявить их с директивой **overload** (перезагрузить)
- В результате станут видны одноименные методы как родителя, так и потомка.

Одноименные методы

Важно:

Чтобы одноименные методы можно было отличить друг от друга, каждый из них **должен иметь уникальный набор параметров**.

- В ходе выполнения программы при обращении к одному из одноименных методов программа проверяет тип и количество фактических параметров обращения и выбирает нужный метод.

Одноименные методы

- При обнаружении одноименного метода компилятор *Delphi* предупреждает о том, что у класса уже есть аналогичный метод с другими параметрами.
- Для подавления сообщений объявление одноименного метода можно сопровождать зарезервированным словом **reintroduce** (вновь ввести).

```
Type
  TForm1 = class (TForm)
     Button1 : TButton;
     Button2 : TButton;
     Button3 : TButton;
     Button4 : TButton;
          Procedure Button1Click(Sender:TObject);
          Procedure Button2Click(Sender:TObject);
          Procedure Button3Click(Sender:TObject);
          Procedure Button4Click(Sender:TObject);
    Private
          procedure Close(S: String); reintroduce; overload;
          procedure Close(I: Integer); reintroduce; overload;
          procedure Close(I, J: Integer); reintroduce; overload;
End;
```

```
implementation

Procedure TForm1.Close(S: String);
    Begin
      Caption := S;
    End;

Procedure TForm1.Close(I: Integer);
    Begin
      Caption := IntToStr(I);
    End;

Procedure TForm1.Close(I, J: Integer);
    Begin
      Caption := IntToStr(i * j);
    End;
```

```
Procedure TForm1.Button1Click (Sender: TObject);
 Begin
   Close('Строка символов');
End;

Procedure TForm1.Button2Click (Sender: TObject);
 Begin
   Close(123);
End;

Procedure TForm1.Button3Click (Sender: TObject);
 Begin
   Close(20, 300);
End;

Procedure TForm1.Button4Click (Sender: TObject);
 Begin
   Close;
End;
```