

Структура класса

Уровни доступа

Уровни доступа класса влияют на то, каким образом он может быть использован.

Классы имеют четыре уровня доступа:

<i>Private</i>	элементы, которые могут быть использованы только в пределах самого класса (не наследуются)
<i>Public</i>	элементы, которые могут быть использованы всюду в программе, в том числе и вне класса
<i>Protected</i>	Элементы, которые могут быть использованы только семейством класса, но которые могут трансформироваться в потомках
<i>Published</i>	элементы, которые могут быть использованы всюду в программе

В Object Pascal уровень доступа к элементам класса может быть описан с помощью одного из следующих четырех ключевых слов: **private**, **public**, **protected** и **published**. Уровень доступа к структурным элементам класса задается во время его объявления с помощью ключевого слова **class**.

```
TVehicle = class  
  private  
  CurrentGear : Integer;  
  Started : Boolean;  
  procedure StartElectricalSystem;  
  procedure StartEngine;  
  protected  
  procedure StartupProcedure;  
  public  
  HaveKey : Boolean;  
  procedure SetGear (Gear : Integer);  
  procedure Brake (Factor : Integer);  
  procedure ShutDown;  
  end;
```

КОНСТРУКТОРЫ

Конструктор — это метод, с помощью которого создаются экземпляры класса. Конструктор используется для инициализации элементов данных класса, выделения дополнительной памяти и других необходимых начальных действий.

В только что приведенном примере класса *TVehicle* конструктор отсутствует.

Включим в класс *TVehicle* объявление конструктора:

```
TVehicle = class
  private
    CurrentGear : Integer;
    Started : Boolean;
    procedure StartElectricalSystem;
    procedure StartEngine;
  protected
    procedure StartupProcedure;
  public
    HaveKey : Boolean;
    procedure SetGear (Gear : Integer);
    procedure Brake (Factor : Integer);
    procedure ShutDown;
    constructor Create; {конструктор}
end;
```

У класса может быть несколько конструкторов.
Конструкторы должны отличаться либо именем, либо количеством и типом параметров либо и тем и другим.

```
TVehicle = class  
{ оставшаяся часть объявления класса }  
    constructor Create;  
    constructor Create(Size : integer);  
    constructor CreateModel(Model : string);  
end;
```

Здесь объявлено три конструктора: два под именем **Create** (отличающиеся набором параметров) и третий - под именем **CreateModel**.


Предположим далее, что есть класс *TMyRect*, инкапсулирующий поведение прямоугольника. Этот класс может имеет сразу несколько конструкторов – объявим два из них.

Объявление класса *TMyRect* выглядит следующим образом:

```
TMyRect = class
private
  Left : Integer;
  Top : Integer;
  Right : Integer;
  Bottom : Integer;
public
  function GetWidth : Integer;
  function GetHeight : Integer;
  procedure SetRect(ALeft, ATop, ARight, ABottom : Integers);
  constructor Create;
  constructor CreateVal (ALeft, ATop, ARight, ABottom : Integers);
end;
```


А вот как должны выглядеть определения конструкторов
(определения помещаются в раздел **implementation**):

```
constructor TMyRect.Create;  
begin  
inherited Create;  
Left := 0;  
Top := 0;  
Right := 0;  
Bottom := 0;  
end;  
constructor TMyRect.CreateVal(ALeft, ATop, ARight, ABottom :Integer);  
begin  
inherited Create;  
Left := ALeft;  
Top := ATop;  
Right := ARight;  
Bottom := ABottom;  
end;
```

Первый конструктор просто инициализирует каждое поле нулевым значением.

Второй присваивает соответствующим полям значения переданных в него параметров.

Обратите внимание на ключе-вое слово **inherited**.

В следующем фрагменте кода происходит создание двух экземпляров класса *TMyRect* — одного с помощью конструктора *Create*, а другого с помощью конструктора *CreateVal*:



```
var  
Rect1 : TMyRect;  
Rect2 : TMyRect;  
begin  
Rect1 := TMyRect.Create;  
Rect2 := TMyRect.CreateVal(0,0, 100, 100);  
end;
```

Память для классов Object Pascal *всегда* выделяется динамически, поэтому все переменные классов, по сути дела, являют-ся указателями.

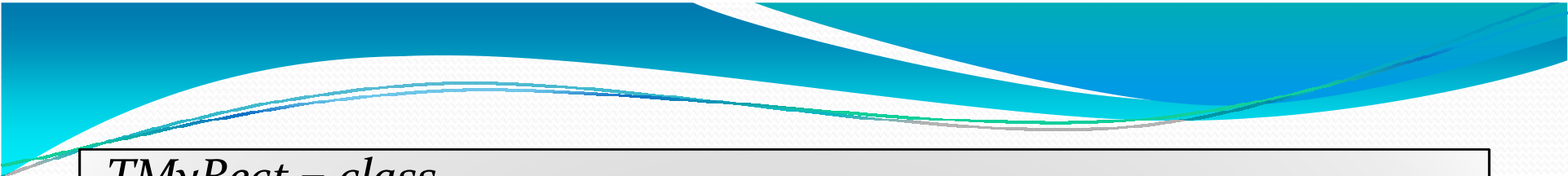
Таким образом, *Rect1* и *Rect2* в предыдущем примере — это указатели на класс TMyRect.

ДЕСТРУКТОР

Деструктор — это специальный метод, автоматически вызываемый перед разрушением объекта. Обычно он используется для освобождения всей выделенной объекту памяти и для других операций, связанных с очисткой.

Класс должен содержать **только один деструктор**, как правило, имеющий имя Destroy, код класса TMyRect, дополненный деструктором.

Ниже приведен (для краткости мы приводим его здесь не полностью): код класса TMyRect, дополненный деструктором.



```
TMyRect = class  
private  
Left : Integer;  
Top : Integer;  
Right : Integer;  
Bottom : Integer;  
Text : PChar; { новое поле }  
public  
function GetWidth : Integer;  
function GetHeight : Integer;  
procedure SetRect(ALeft, ATop, ARight., ABottom : Integer) ;  
constructor Create;  
constructor CreateVal(ALeft, ATop, ARight, ABottom : Integer);  
destructor Destroy; override;  
end;
```



```
constructor TMyRect.Create;
```

```
begin
```

```
inherited Create;
```

```
{ Выделяем память для ограниченной нулем строки. }
```

```
Text := AllocMem(1024) ;
```

```
end;
```

```
destructor TMyRect.Destroy;
```

```
begin
```

```
{ Освобождаем выделенную память. }
```

```
FreeMem(Text);
```

```
inherited Destroy;
```

```
end;
```

В конструкторе класса *TMyRect* происходит динамическое выделение памяти для строки с ограничивающим нулем *Text* (типа *PChar*), тогда как в деструкторе эта память освобождается.

Обратите внимание на ключевое слово **override**.

ПОЛЯ ДАННЫХ

Поля данных — это переменные, объявленные в объявлении класса; можно рассматривать их как переменные, имеющие своей областью действия класс.

Давайте снова вернемся к классу TMyRect. В нем нет открытых полей. Попытавшись выполнить приведенный ниже код, появляется сообщение об ошибке времени компиляции Undeclared identifier: 'Left' (необъявленный идентификатор: 'Left'):

```
Rect := TMyRect.CreateVal(0, 0, 100, 100);
```

```
Rect.Left := 20; ( ошибка компиляции! }
```

МЕТОДЫ

Методы — это процедуры и функции, принадлежащие классу. Они локальны по отношению к своему классу; их как бы не существует за его пределами.

Методы могут быть вызваны только внутри самого класса или при посредстве его объектов.

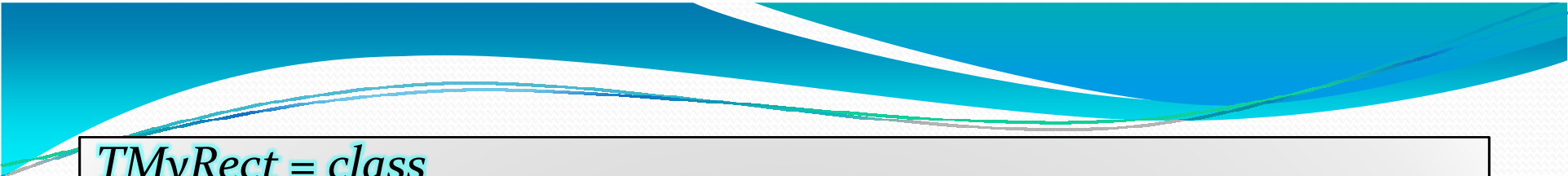
Любой обработчик события является методом класса.

1. **Открытые** (или общедоступные) методы, вместе со свойствами, представляют собой пользовательский интерфейс класса.
2. **Закрытые** методы предназначены для внутреннего использования классом и не должны вызываться его пользователями
3. **Защищенными** называются методы, доступные только для классов, производных от данного

УКАЗАТЕЛЬ SELF

У каждого класса есть скрытое поле `Self`, явно не определенное среди полей класса, которое представляет собой указатель на соответствующий экземпляр класса.

Очевидно, что это определение требует некоторых пояснений. Сначала давайте посмотрим, как выглядело бы объявление класса `TMyRect`, если бы поле `Self` не было скрытым:



```
TMyRect = class  
private  
Self : TMyRect;  
Left : Integer;  
Top : Integer;  
Right : Integer;  
Bottom : Integer;  
Text : PChar;  
public  
function GetWidth : Integer;  
function GetHeight : Integer;  
procedure SetRect(ALeft, ATop, ARight, ABottom : Integer);  
constructor Create;  
constructor CreateVal(ALeft, ATop, ARight, ABottom : Integer);  
destructor Destroy; override;  
end;
```

Именно так класс TMyRect выглядит с точки зрения компилятора. При создании в памяти объекта класса указатель Self автоматически инициализируется адресом этого объекта:

Rect := TMyRect.CreateVal (0, 0, 100, 100);

{ Теперь 'Rect' и 'Rect.Self' имеют одно и то же значение, }

{ поскольку оба содержат адрес объекта в памяти. }

Каждый экземпляр класса имеет собственную копию полей данных

Добавим к классу TMyRect следующее определение функции GetWidth:

```
function TMyRect.GetWidth : Integer;  
begin  
Result := Right - Left;  
end;
```

Но так эта функция выглядит для нас с вами.

Компилятор же видит ее несколько иначе:

```
function TMyRect.GetWidth : Integer;  
begin  
Result := Self.Right - Self.Left;  
end ;
```