

Глава 11. ОБЪЕКТЫ

- Описание классов, создание объектов
- Ограничение доступа к полям и методам
- Создание библиотек классов
- Реализация принципа наследования
- Полиморфизм
- Раннее связывание
- Позднее связывание, виртуальные методы

ОПИСАНИЕ КЛАССА

Класс – это структурный тип данных, который включает описание полей данных, а также процедур и функций (методы), работающих с этими полями.

Type

```
PMouse = ^TMouse;  
TMouse = object      {class}  
    Visible : Boolean; {признак включен/выключен}  
    Keys : Byte;       {количество кнопок}  
    constructor Init;  {конструктор}  
    destructor Done;   {деструктор}  
    procedure On;      {включить указатель мыши}  
    procedure Off;     {выключить указатель мыши}  
end;;
```

Метод Init создает и инициализирует объект – конструктор (описание начинается со слова constructor). Любой класс должен иметь как минимум один конструктор (если у класса есть виртуальные методы).

Обратным конструктору является метод уничтожения объекта Done – деструктор (описывается словом destructor). Класс не может иметь более одного деструктора.

Если объект динамический, то конструктор выделяет память для него в куче, а деструктор – освобождает память.

СОЗДАНИЕ ОБЪЕКТА

Объект – это экземпляр класса (переменная типа object или class). Подобно записи, объект используется с префиксной частью в виде имени этого объекта.

Статический объект:

```
Var
    Mouse : TMouse;
Begin
    Mouse.Init;      {инициализация объекта}
    Mouse.On;        {включить указатель мыши}
    ...
End.
```

Динамический объект:

```
Var
    Mouse : PMouse;      {тип – указатель на TMouse}
Begin
    New(Mouse, Init);    {инициализация объекта}
    Mouse^.On;           {включить указатель мыши}
    ...
    Dispose(Mouse, Done); {уничтожить объект}
End.
```

```
if Mouse^.Keys > 0 then
begin
    Mouse^.On;
    ...
    Mouse^.Off;
end;
```

Фрагмент ошибочен с точки зрения объектного подхода, поскольку нарушен принцип **инкапсуляции**. Получать информацию об объекте следует только при помощи операции селектора, т.е. специальным методом.

ОГРАНИЧЕНИЕ ДОСТУПА К ПОЛЯМ И МЕТОДАМ

Type

```
PMouse = ^TMouse;  
TMouse = object  
    constructor Init;  
    destructor Done;  
    procedure On;  
    procedure Off;  
    function GetKeys : Byte; {число кнопок у мыши}  
    function IsVisible : Boolean; {видна мышь или нет}  
private  
    Visible : Boolean;  
    Keys : Byte;  
end;
```

Введены два метода-селектора GetKeys, IsVisible для определения состояния объекта.

Сами поля Visible и Keys убраны в специальную частную секцию private. Эта секция делает недоступными для непосредственного использования в программе поля и методы, расположенные в ней.

БИБЛИОТЕКИ КЛАССОВ

При создании библиотек классов они описываются в отдельных модулях, которые могут подключаться к основной программе для использования этих классов. Для соблюдения принципа инкапсуляции, класс описывают в разделе Interface, а его методы – в разделе Implementation, что позволяет "скрыть" внутреннее содержание методов.

```
Unit UMouse;
Interface      {секция интерфейса}
Type
    PMouse = ^TMouse;
    TMouse = object
        constructor Init;
        destructor Done;
        procedure On;
        procedure Off;
        function GetKeys : Byte;
        function IsVisible : Boolean;
    private
        Visible : Boolean;
        Keys : Byte;
    end;
```

```

Implementation      {секция реализации}
  constructor TMouse.Init;
    begin
      ...
    end;
  ...
  function TMouse.IsVisible : Boolean;
    begin
      ...
    end;
  ...
End.

```

```

Program Example;
  Uses UMouse;      {подключение модуля с классом}
  Var Mouse : PMouse;
  Begin
    ...
  End.

```

НАСЛЕДОВАНИЕ

При наследовании объекты класса-потомка получают возможность использования ("наследуют") полей и методов класса-родителя, что позволяет повторно не определять эти компоненты класса.

```
Program NewMouseDemo ;
```

```
  Uses UMouse;    {подключаем созданный ранее модуль}
```

```
  Type
```

```
    PNewMouse = ^TNewMouse;
```

```
    TNewMouse = object(TMouse)    {предок - TMouse}
```

```
      function GetClick(Var X,Y : Word) : Byte;
```

```
    end;
```

```
  Var
```

```
    Mouse : PNewMouse;
```

```
    A,B : Word;
```

```
    Button : Byte;
```

```
  Function TNewMouse.GetClick(Var X,Y : Word) : Byte;
```

```
    begin                                {функция возвращает номер нажатой
```

```
      ...                                кнопки в своем значении и текущие
```

```
    end;                                координаты мыши в параметрах-
```

```
      переменных} ;
```


НАСЛЕДОВАНИЕ

Begin

New(Mouse, Init) ;

if Mouse^.GetKeys > 0 **then**

begin {проверка, подключена ли мышь}

Mouse^.On;

repeat

Button := Mouse^.GetClick(A,B) ;

if Button > 0 **then**

Writeln(Button,A,B) ;

until Button = 3;

Mouse^.Off;

end;

Dispose(Mouse, Done) ;

End.

Новый класс TMouseNew унаследовал все свойства класса TMouse, т.к. является его потомком. Соответственно, класс TMouse для класса TMouseNew является предком, или родителем. В новом классе описан только один новый метод GetClick для получения информации о нажатой кнопке. При этом все методы класса TMouse становятся доступными для экземпляров класса TMouseNew.

Var M1 : PMouse; M2 : PNewMouse; ... M1 := M2 {допустимо}

ПОЛИМОРФИЗМ

В объектно-ориентированном программировании полиморфизм проявляется в возможности переопределять методы класса-предка, т.е. расширять свойства объекта путем "перестраивания" его методов.

Type

```
PPoint = ^TPoint;           {объект - точка на экране}
TPoint = object
    X, Y : Word;
    Color : Byte;
    constructor Init (InitX, InitY : Word;
                     InitColor : Byte);

    destructor Done;
    procedure Draw;
end;
PCircle = ^TCircle; {объект - окружность на экране}
TCircle = object(TPoint) {производный от TPoint}
    Radius : Word;
    constructor Init (InitX,InitY,InitRadius:Word;
                     InitColor : Byte);

    procedure Draw;
end;
```

ПОЛИМОРФИЗМ

```
Constructor TPoint.Init;      {при создании объекта
begin                          задаются координаты точки и
    X := InitX;                ее цвет через параметры
    Y := InitY;                конструктора}
    Color := InitColor;
end;

Destructor TPoint.Done;      {нет операторов, объект
begin end;                    уничтожается с помощью Dispose}

Procedure TPoint.Draw;
begin
    Writeln('Рисуем точку x,y=',x,y,'Цвет=',Color);
end;

Constructor TCircle.Init;
begin
    Radius := InitRadius;
    TPoint.Init(InitX,InitY,InitColor);
end;

Procedure TCircle.Draw;
begin
    Writeln('Рисуем окружность x,y,R=',x,y,Radius,
            'Цвет=',Color);
end;
```

ПОЛИМОРФИЗМ

В программе описаны два класса: TCircle и его потомок TPoint. Деструктор Done наследуется, а методы Init и Draw на основе полиморфизма перекрываются.

В конструкторе класса TCircle не только инициализируется новое поле Radius, но и вызывается в явном виде конструктор "родителя". Это делается для того, чтобы проинициализировать те поля, которые "перешли по наследству" этому классу.

Метод Draw класса TCircle описан заново – рисование окружности отличается от рисования точки.

Приведенный пример относится к случаю **простого полиморфизма**, когда при вызове переопределенного метода тип объекта (класс), для которого вызывается данный метод, точно известен. При этом адреса методов-процедур, которые ассоциируются с именем Draw, и где размещаются команды этих методов, подставляются в вызовы данных методов на этапе компиляции. Такой процесс называется **ранним связыванием** – т.е. включением в машинный код точных адресов на те точки программы, с которых начинается размещение кода метода (такой метод называется **статическим**).

Var

Point : PPoint; Circle : PCircle;

Begin

... Point^.Draw; Circle^.Draw; ...

End.

ПОЛИМОРФИЗМ

Действия компилятора при обработке статических методов объектов, связанных иерархически:

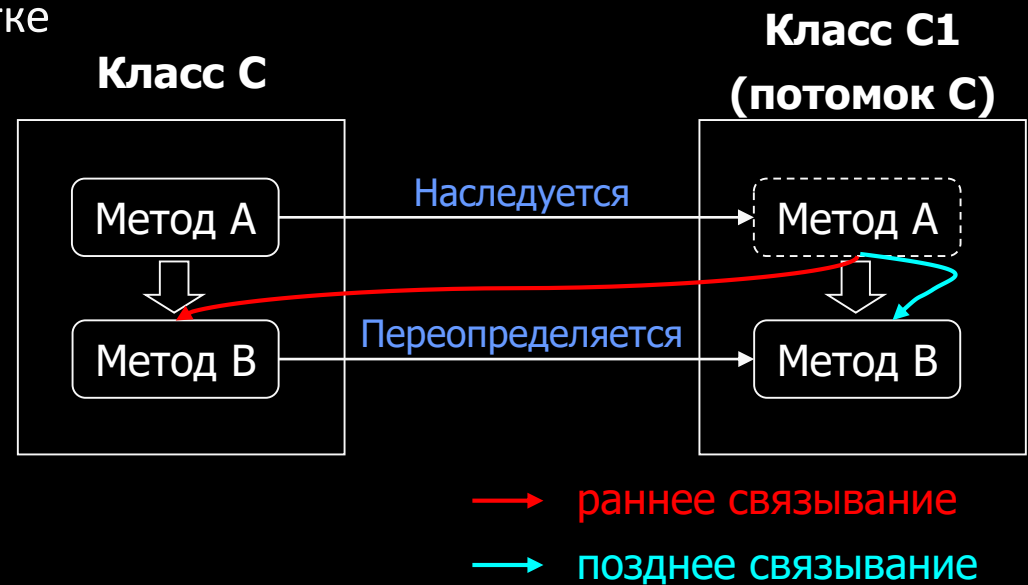
1. При вызове метода компилятор устанавливает тип объекта, вызывающего метод.

2. Установив тип, компилятор ищет метод в пределах типа объекта.

Найдя его, компилятор назначает вызов этого метода.

3. Если метод не найден, то компилятор начинает рассматривать тип непосредственного прародителя и ищет метод, имя которого вызвано, в пределах родительского типа. Если он найден, то вызов заменяется на вызов метода родителя. Если нет, то компилятор "поднимается" еще на один уровень и т.д. При этом, если метод родителя вызывает другие методы, то последние также будут методами родителя, даже если потомки имеют свои собственные методы.

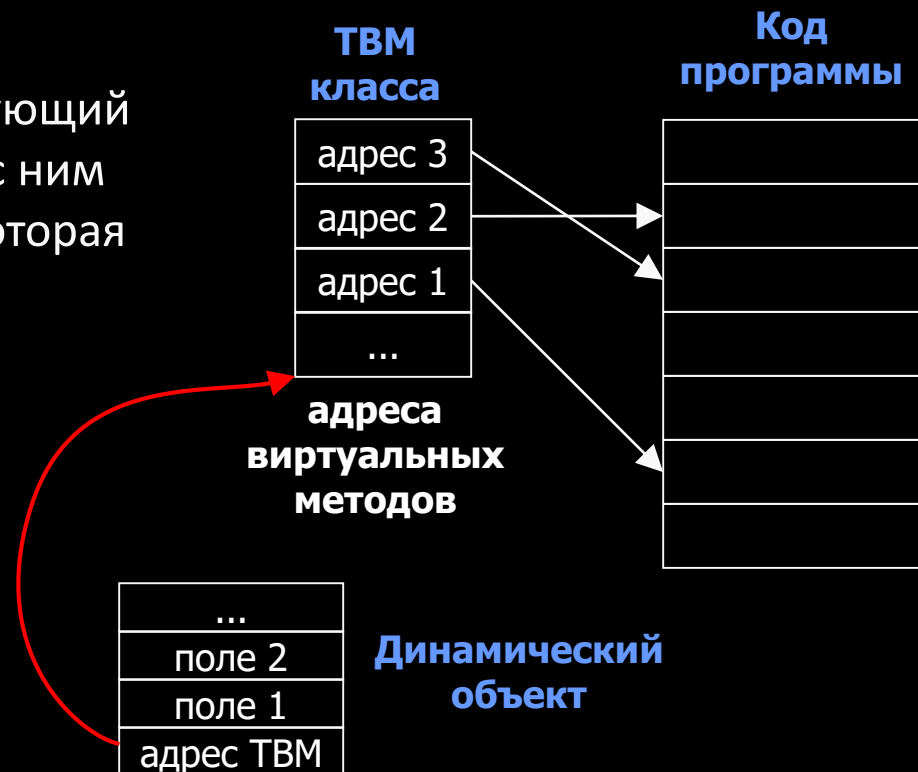
Для стандартизации работы с объектами, имеющими полиморфные методы, введено понятие **позднего связывания**, когда компилятор формирует косвенное обращение к адресу ТВМ. Реальный адрес нужного метода не известен до момента выполнения программы. Этот метод подключается в процессе выполнения программы, когда однозначно определен тип объекта (класс).



ПОЛИМОРФИЗМ

Замещаемые (полиморфные) методы объектов, для которых необходимо реализовать механизм позднего связывания, называют **виртуальными** и отмечают в описании объекта стандартной директивой `virtual`. Одни и те же виртуальные методы должны иметь одинаковые заголовки у всех объектов данной иерархии. Метод, объявленный виртуальным в некотором классе, обязан быть виртуальным и во всех его наследниках. Использование механизмов позднего связывания возможно как для статических, так и динамических объектов (размещенных в динамической памяти).

Каждый класс, содержащий или наследующий виртуальные методы, имеет связанную с ним **таблицу виртуальных методов** (ТВМ), которая размещается в статической памяти. В этой таблице содержится, в частности, информация об явных адресах всех виртуальных методов данного класса. При создании объекта (после выполнения метода-конструктора) в него добавляется специальное поле – указатель на ТВМ данного класса.



Type

```

PPoint = ^TPoint;
TPoint = object
    X, Y : Word;
    Color : Byte;
    constructor Init (InitX, InitY : Word;
                     InitColor : Byte);

    destructor Done;
    procedure Draw; virtual;
end;
PCircle = ^TCircle;
TCircle = object(TPoint)
    Radius : Word;
    constructor Init (InitX,InitY,InitRadius:Word;
                     InitColor : Byte);

    procedure Draw; virtual;
end;
...
{описания методов - прежние}
...
```

Var

```
Point : PPoint;
Circle : PCircle;
P : array[1..2] of PPoint;
i : Byte;
```

Begin

```
New(Point, Init(...));
New(Circle, Init(...));
P[1] := Point;
P[2] := Circle;
for i := 1 to 2 do P[i]^Draw;
```

End.

Point^

| |
|-----------|
| X |
| Y |
| Color |
| Адрес TBM |

**TBM
класса
TPoint**

| |
|------|
| Draw |
| ... |

Circle^

| |
|-----------|
| X |
| Y |
| Color |
| Radius |
| Адрес TBM |

**TBM
класса
TCircle**

| |
|------|
| Draw |
| ... |

Тип обеих переменных Point и Circle – указатель, поэтому можно ввести массив указателей. Присваивание P[2] := Circle корректно с точки зрения совместимости типов – все объекты-потомки совместимы со своими предками (обратное не верно).

Если бы метод Draw был статическим, то для рисования окружности потребовался бы, например, вызов: PCircle(P[2])^Draw

Удобство использования виртуальных методов становится более заметно при увеличении числа и усложнении иерархии объектов.