

2021

Часть 2 Объектно-ориентированное программирование

МГТУ им. Н.Э. Баумана

Факультет Информатика и системы управления

Кафедра Компьютерные системы и сети

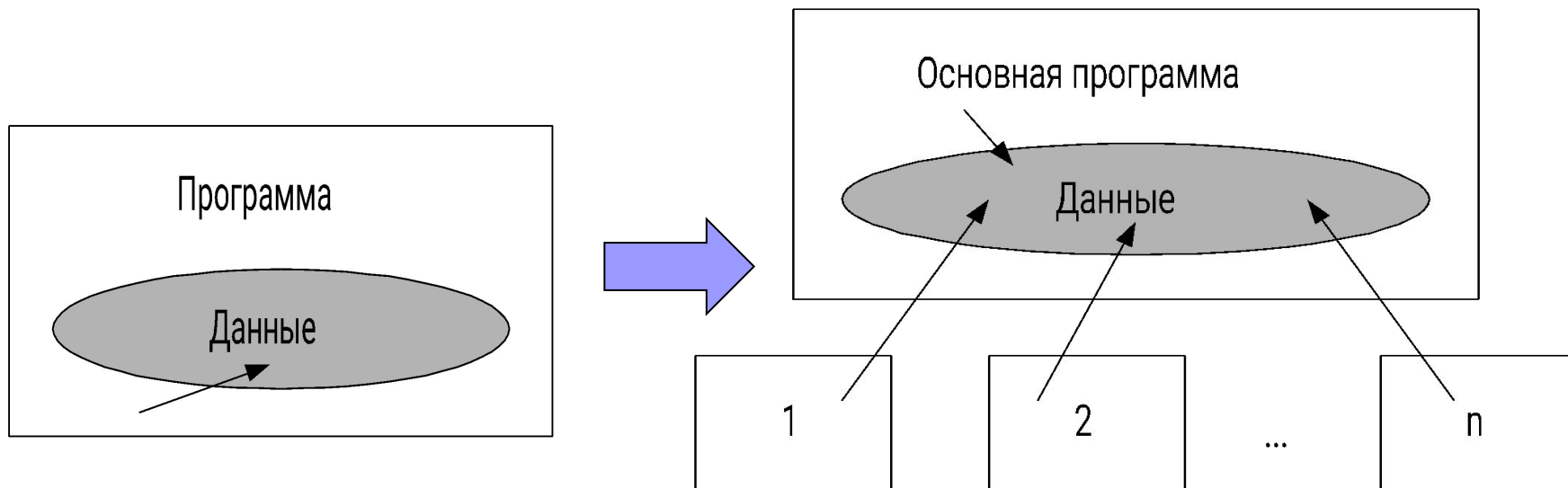
Лектор: д.т.н., проф.

Иванова Галина Сергеевна

Введение. Эволюция технологии разработки ПО.

Процедурная и объектная декомпозиция

1. «Стихийное» программирование – до середины 60-х годов XX века – технология отсутствует – программирование – искусство создания программ – в конце периода появляется возможность создания подпрограмм – используется процедурная декомпозиция.

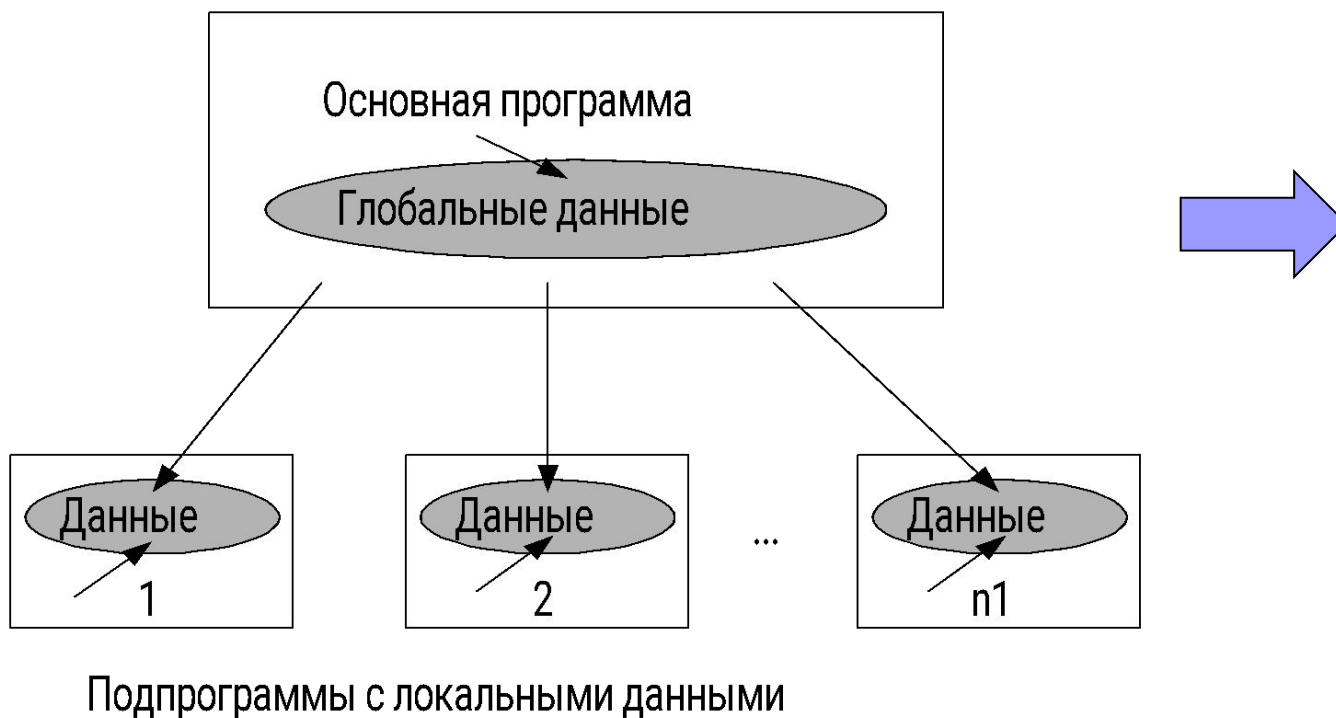


Слабое место – большая вероятность испортить глобальные данные.

Эволюция технологии разработки ПО (2)

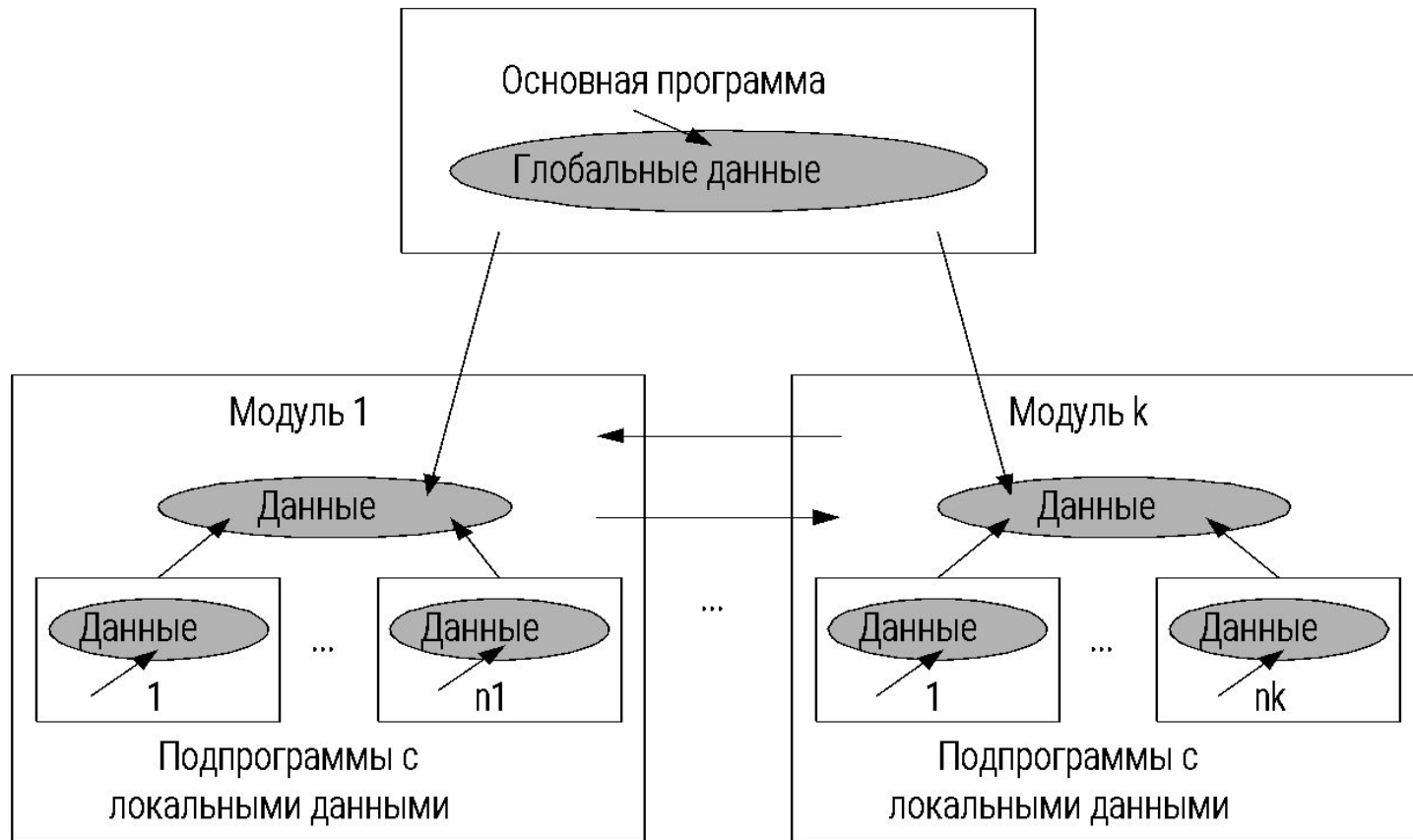
2. Структурный подход к программированию - 60-70-е годы XX века – технология, представляющая собой набор рекомендаций и методов, базирующихся на большом опыте работы:

- нисходящая разработка;
- декомпозиция методом пошаговой детализации;
- структурное программирование;
- сквозной структурный контроль и т. д.



Эволюция технологии разработки ПО (3)

Модульное программирование – выделение групп подпрограмм, использующих общие глобальные данные в модули – отдельно компилируемые части программы (многоуровневая декомпозиция).

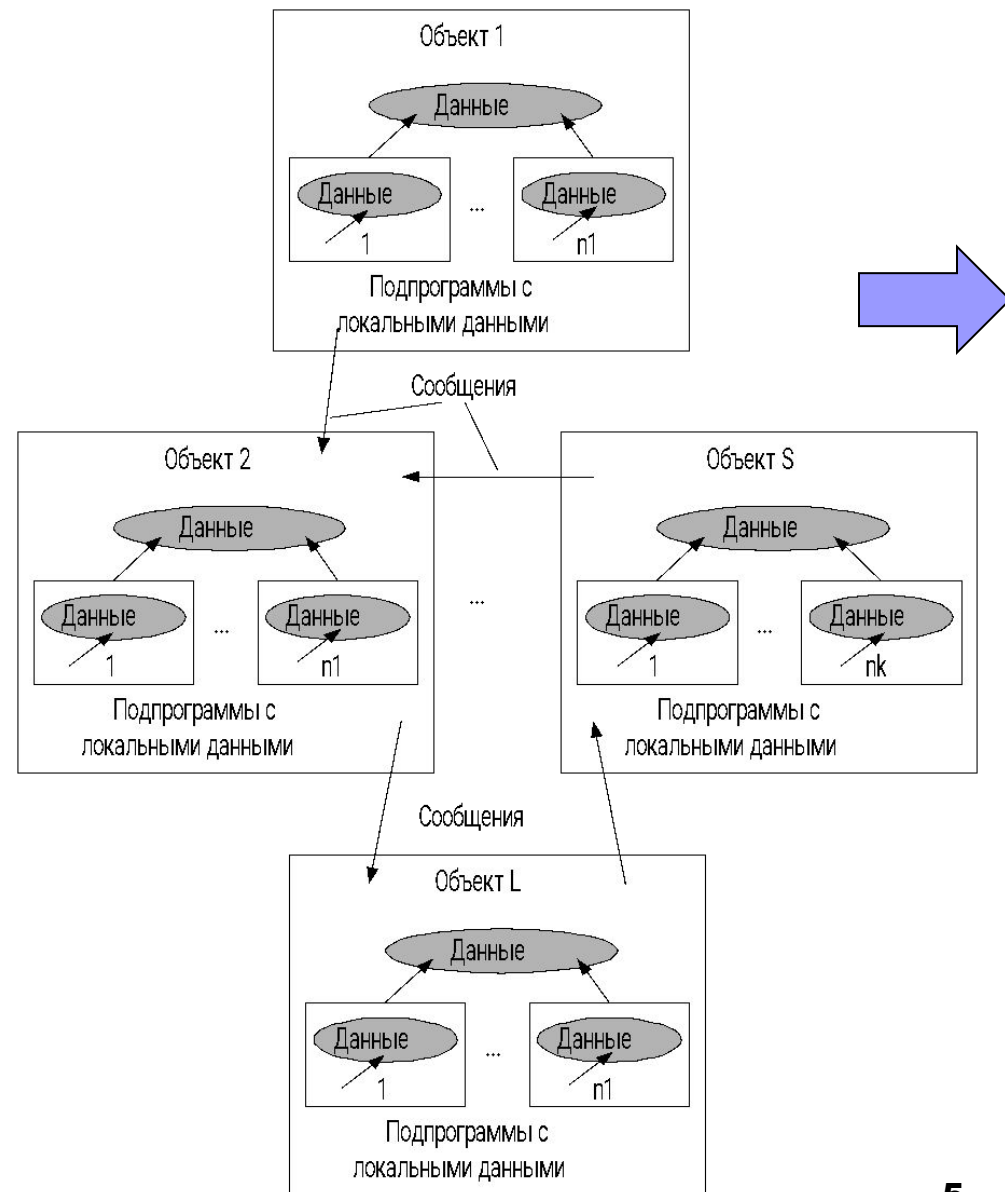


Слабое место – большое количество передаваемых параметров.

Эволюция технологии разработки ПО (4)

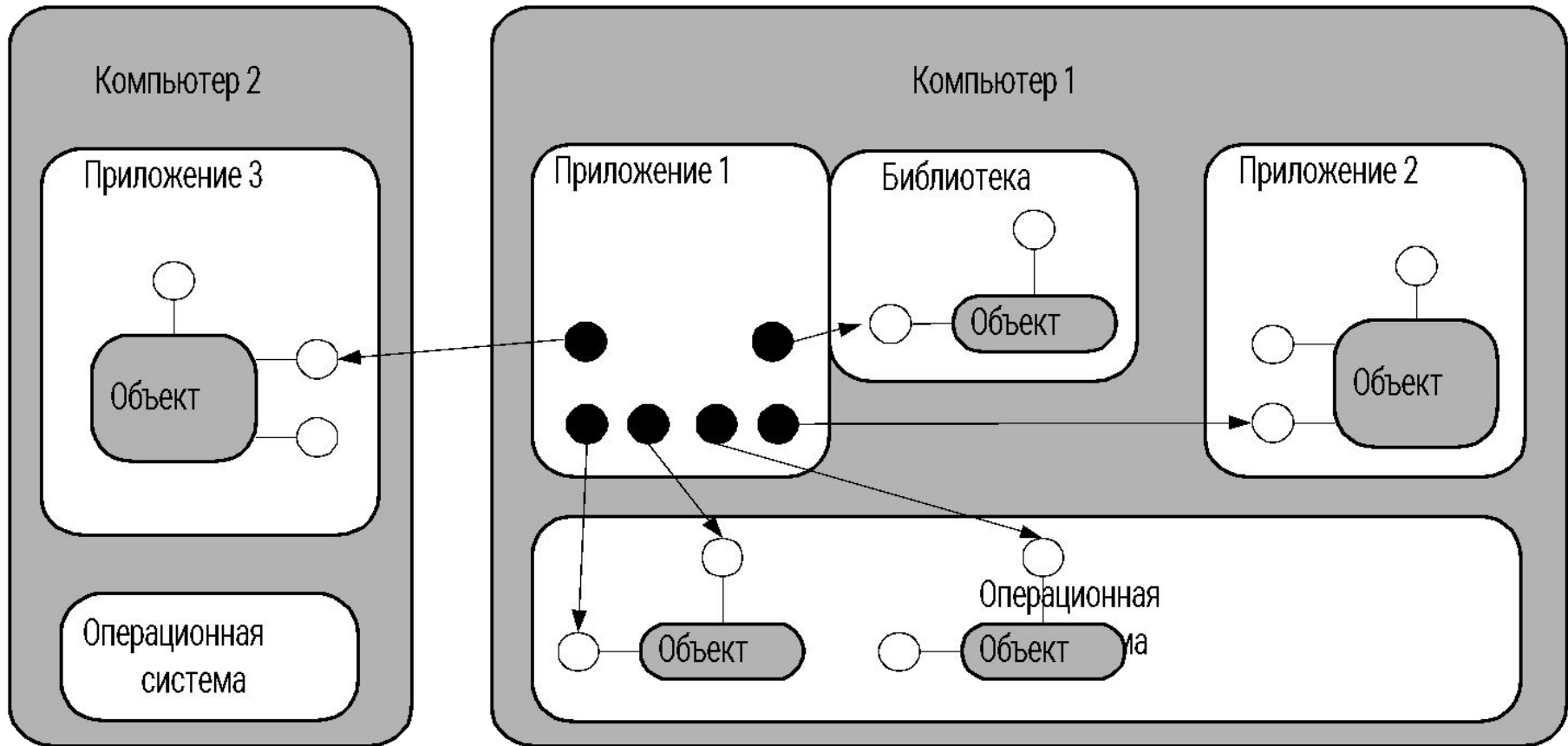
3. **Объектный подход к программированию** – с середины 80-х до наших дней.

Объектно-ориентированное программирование – технология создания сложного программного обеспечения, основанная на представлении программы в виде системы объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств.



Эволюция технологии разработки ПО (5)

Компонентный подход – с конца 90-х годов XX века (COM-технология, Corba, SOAP) – подключение объектов через универсальные интерфейсы – развитие сетевого программирования – появление CASE-технологий.



Пример

Разработать программную систему, которая для указанной функции на заданном отрезке:

- строит таблицу значений с определенным шагом;
- определяет корни;
- определяет максимум и минимум.

Формы интерфейса пользователя

Программа исследования функций.

Введите функцию или слово «Конец»: $y = \cos(x) - 1$

Назначьте интервал: $[-1, 0)$

Введите номер решаемой задачи

(1 - построение таблицы значений;

2 - нахождение корней;

3 - нахождение минимума и максимума;

4 - смена функции или завершение программы) : 1

Построение таблицы.

Введите шаг: 0.01

Таблица значений:

x=	y=
----	----

...

Нахождение корней.

Таблица корней:

x=	y=
----	----

...

Экстремумы.

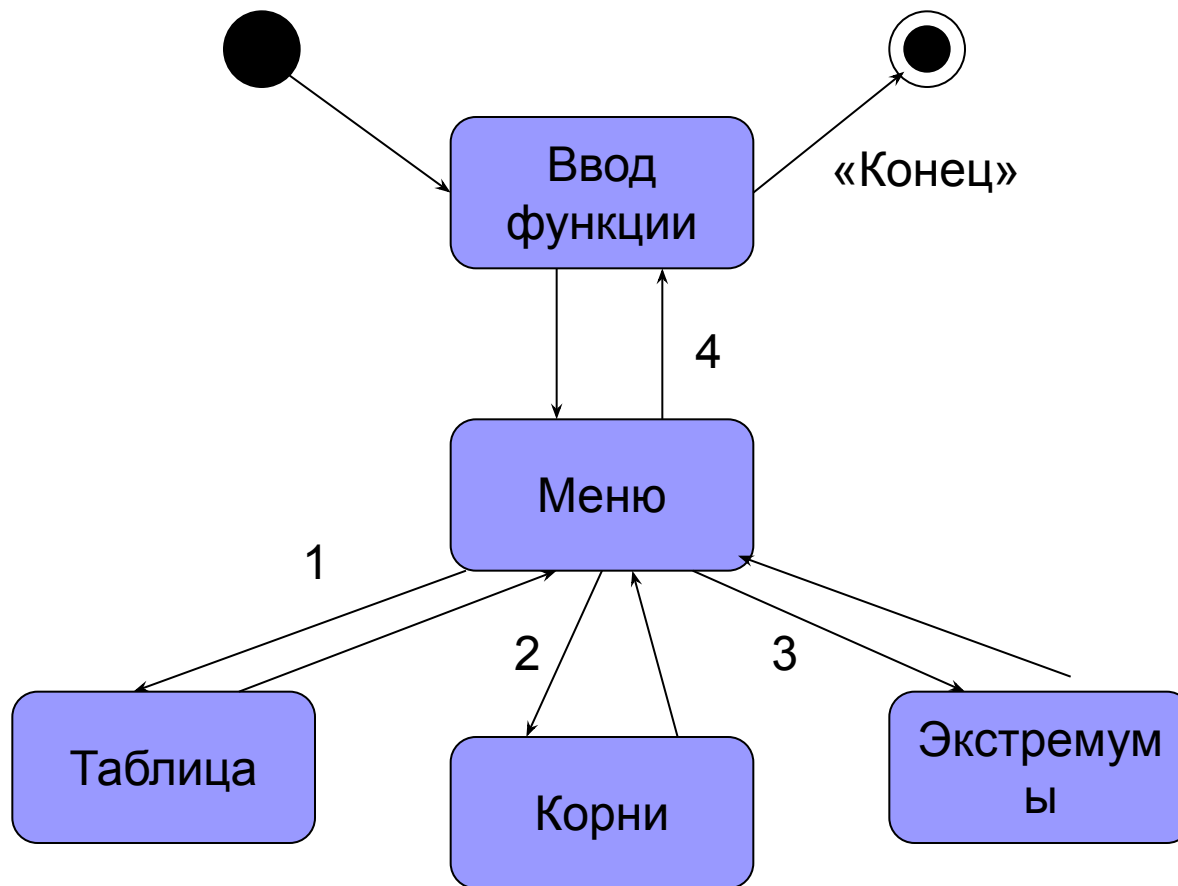
Минимум:

x=	y=
----	----

Максимум:

x=	y=
----	----

Диаграмма состояний интерфейса пользователя



Разработка схем алгоритмов методом пошаговой детализации

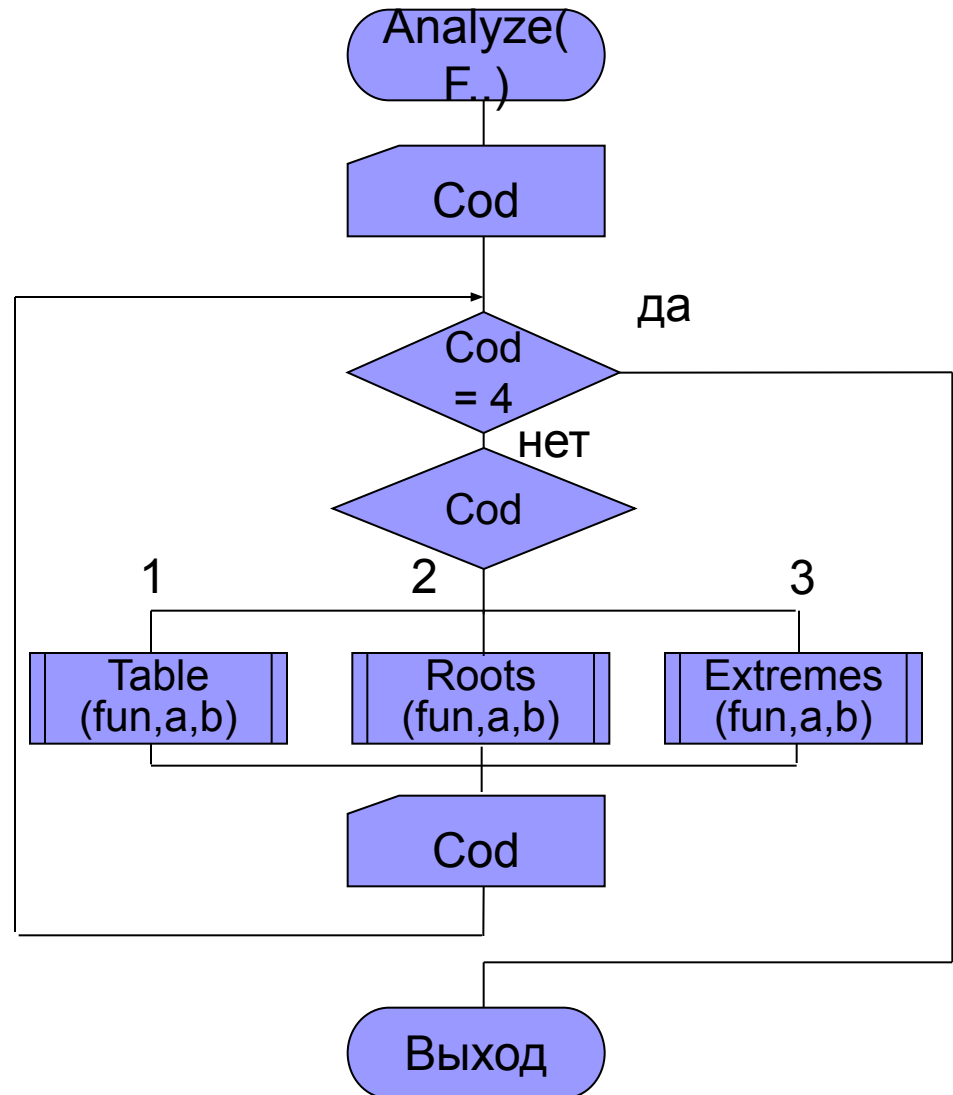
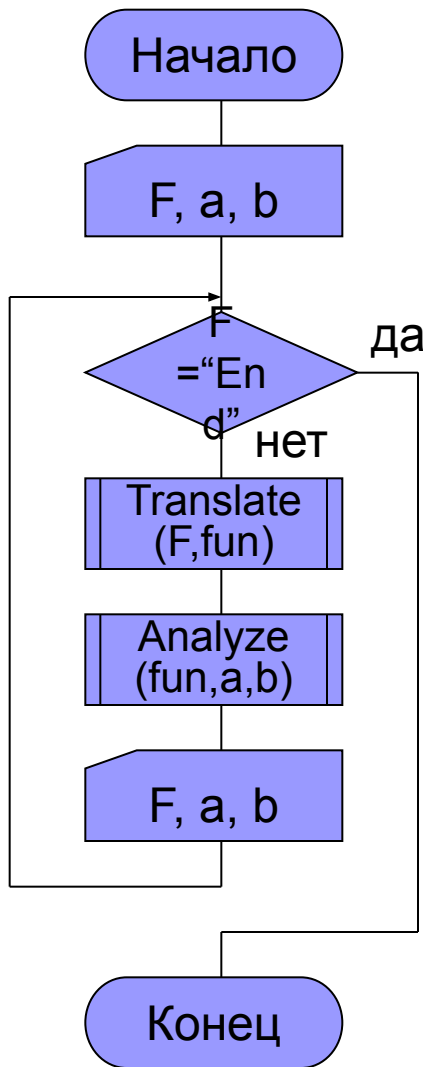
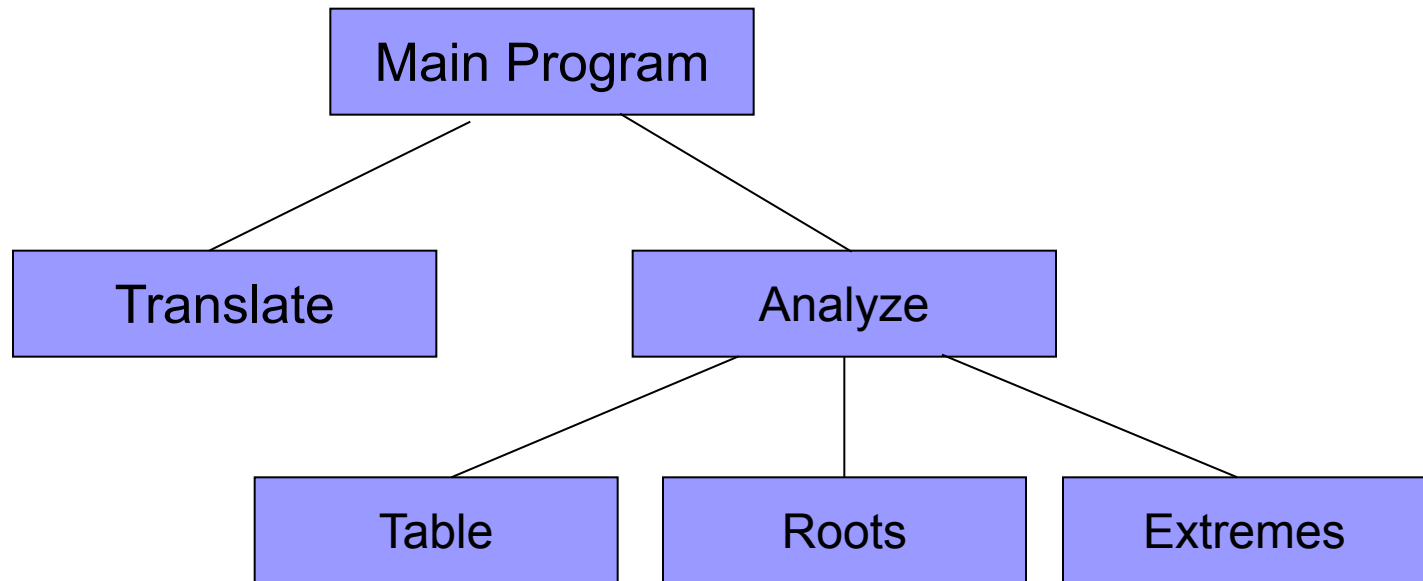


Схема структурная программы



Процедурная декомпозиция – процесс разбиения программы на подпрограммы.

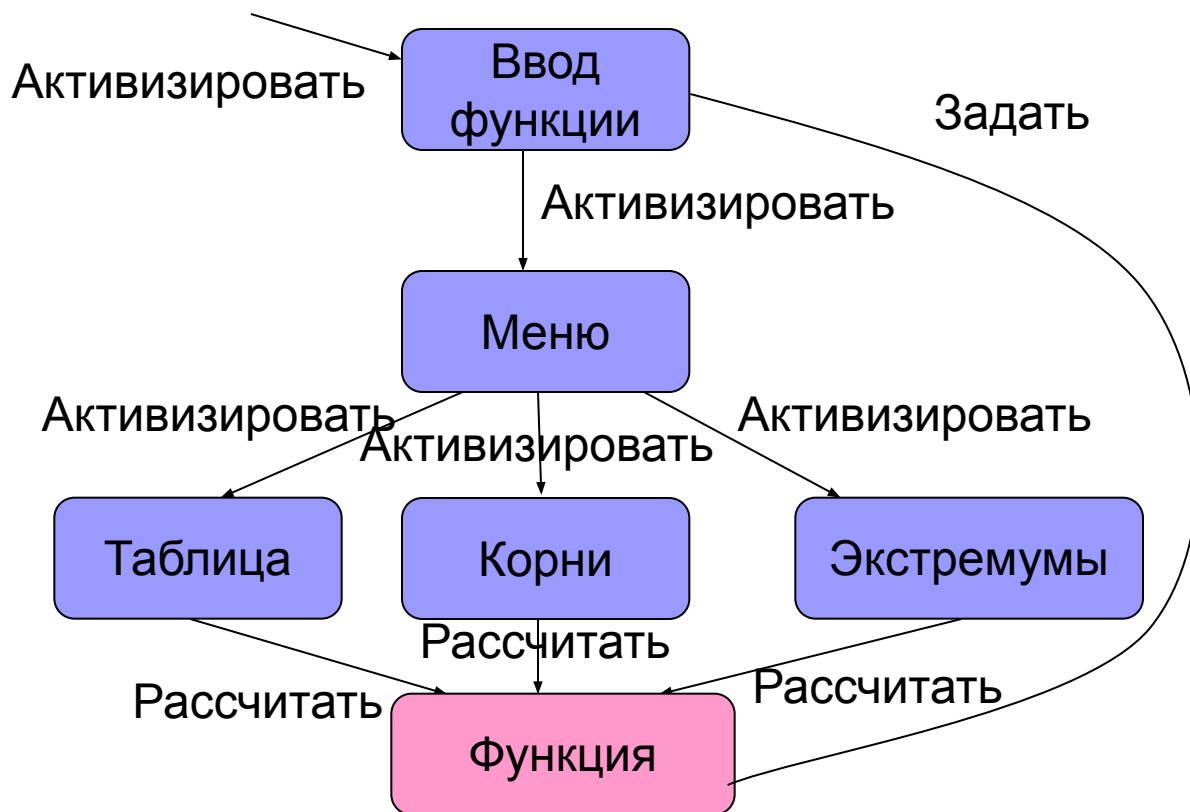
Структурной называют декомпозицию, если:

- каждая подпрограмма имеет один вход и один выход;
- подпрограммы нижних уровней не вызывают подпрограмм верхних уровней;
- размер подпрограммы не превышает 40-50 операторов;
- в алгоритме использованы только структурные конструкции.

Объектная декомпозиция

Объектная декомпозиция – процесс представления предметной области задачи в виде отдельных функциональных элементов (объектов предметной области), обменивающихся в процессе выполнения программы входными воздействиями (сообщениями).

Объект отвечает за выполнение некоторых действий, инициируемых сообщениями и зависящих от параметров объекта.



Объект предметной области характеризуется:

- именем;
- состоянием;
- поведением.

Состояние – совокупность значений характеристик объекта, существенных с т. з. решаемой задачи.

Поведение – совокупность реакций на сообщения.

Реализация объектов предметной области



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

Применительно к классам такие процедуры и функции получили название **методов**.

Объект-переменная – переменная типа «класс».

Основные принципы, на которых базируется объектно-ориентированное программирование

- **абстрагирование** – выделение абстракций в предметной области задачи; под **абстракцией** при этом понимается совокупность существенных характеристик некоторого объекта предметной области, которые отличают его от всех других видов объектов;
- **инкапсуляция** – размещение в одном программном компоненте (объекте) данных и подпрограмм, которые с этими данными работают;
- **ограничение доступа** – сокрытие отдельных элементов реализации абстракции, не затрагивающих существенных характеристик ее как целого;
- **модульность** – принцип разработки программной системы, предполагающий реализацию ее в виде отдельных частей;
- **иерархичность** – использование иерархий при разработке программных систем; при этом используются как иерархии "целое-часть", так и иерархии "общее-частное";
- **типизация** – ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов (или сильно сужающее возможность такой замены).

Методы построения классов

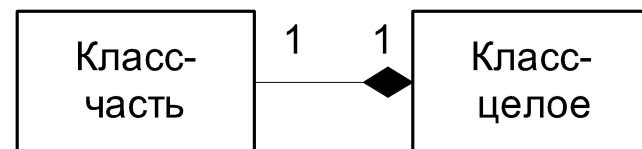
1. **Наследование** – механизм, позволяющий строить классы на базе более простого посредством добавления полей и определения новых методов.

При этом исходный класс, на базе которого выполняется построение, называют *родительским* или *базовым*, а строящийся класс – *потомком* или *производным* классом.

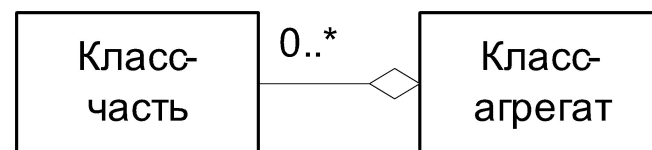
Если при наследовании какие-либо методы переопределяются, то такое наследование называется *полиморфным*.



2. **Композиция** – механизм, позволяющий включать один или несколько объектов других классов в объекты конструируемого.



3. **Наполнение** – механизм, позволяющих включать или не включать объекты других классов в объект конструируемого..



Глава 7 Средства объектно- ориентированного программирования

МГТУ им. Н.Э. Баумана

Факультет Информатика и системы
управления

Кафедра Компьютерные системы и сети

Лектор: д.т.н., проф.

Иванова Галина Сергеевна

7.1 Определение класса, инкапсуляция полей и методов класса. Объявление объектов и инициализация полей

С точки зрения синтаксиса **класс** – структурный тип данных, в котором помимо полей разрешается описывать **прототипы** (заголовки) процедур и функций, работающих с этими полями данных.



```
Type TRoom = object
    length, width: single;
    function Square: single; {прототип функции}
end;
```

```
Function TRoom.Square;
Begin
    Result := length*width;
End;
```

Поскольку данные и методы **инкапсулированы** в пределах класса, все поля автоматически доступны из любого метода

Неявный параметр Self

Любой метод неявно получает параметр **Self** – ссылку (адрес) на поля объекта, и обращение к полям происходит через это имя.

```
Function TRoom.Square;  
    Begin  
        Result:= Self.length* Self.width;  
    End;
```

При необходимости эту ссылку можно указывать явно:

@Self – адрес области полей данных объекта.

Объявление объектов класса

Примеры:

```
Var A:TRoom;      {объект A класса TRoom}  
    B:array[1..5] of TRoom; {массив объектов типа TRoom}  
Type pTRoom=^TRoom; {тип указателя на объекты класса TRoom}  
Var  pC: pTRoom;   {указатель на объекта класса TRoom}
```

Для динамического объекта необходимо выделить память:

```
New (pC) ;
```

а после его использования – освободить память:

```
Dispose (pC) ;
```

Доступ к полям и методам аналогичен доступу к полям записей:

Примеры:

а) `v:=A.length;`

б) `s:= A.Square;`

в) `s:=s+B[i].Square;`

г) `pC^.length:=3;`

Инициализация полей прямой записью в поле

```
Program Ex_7_01a;  
{ $APPTYPE CONSOLE }  
Uses SysUtils;  
Type TRoom = object  
    length, width:single;  
    function Square:single;  
end;  
Function TRoom.Square;  
    Begin  
        Result:= length* width;  
    End;  
Var A:TRoom;  
Begin  
    A.length:=3.5;  
    A.width:=5.1;  
    WriteLn('S = ',A.Square:8:3);  
    ReadLn;  
End.
```

Инициализация при объявлении объекта

```
Program Ex_07_01b;  
{ $APPTYPE CONSOLE }  
Uses SysUtils;  
Type TRoom = object  
    length, width:single;  
    function Square:single;  
end;  
Function TRoom.Square;  
    Begin  
        Result:= length* width;  
    End;  
Var A:TRoom = (length:3.5; width:5.1);  
Begin  
    WriteLn('S= ',A.Square:8:3);  
    ReadLn;  
End.
```

Инициализация посредством метода

```
Program Ex_07_01c;  
{$APPTYPE CONSOLE}  
Uses SysUtils;  
Type TRoom = object  
    length, width:single;  
    function Square:single;  
    procedure Init(l,w:single);  
end;  
Function TRoom.Square;  
    Begin Square:= length*width; End;  
Procedure TRoom.Init;  
    Begin length:=l; width:=w; End;  
Var A:TRoom;  
Begin  
    A.Init(3.5,5.1);  
    WriteLn('S= ',A.Square:8:3);  
    ReadLn;  
End.
```

Операция присваивания объектов

Над объектами одного класса определена операция **присваивания**.
Физически при этом происходит копирование полей одного объекта в другой методом «поле за полем»:

Пример:

```
Var A:TRoom =(length:3.7; width:5.2) ;  
Var B:TRoom ;  
...  
B:=A;
```

7.2 Ограничение доступа к полям и методам

Ограничение в пределах модуля, а не в пределах класса!

```
Unit Room;
```

```
Interface
```

```
    Type TRoom = object
```

```
        private length, width: single;
```

```
        public function Square: single;
```

```
        procedure Init(l,w: single);
```

```
    end;
```

```
Implementation
```

```
    Function TRoom.Square;
```

```
        Begin Result:= length* width; End;
```

```
    Procedure TRoom.Init;
```

```
        Begin length:=l; width:=w; End;
```

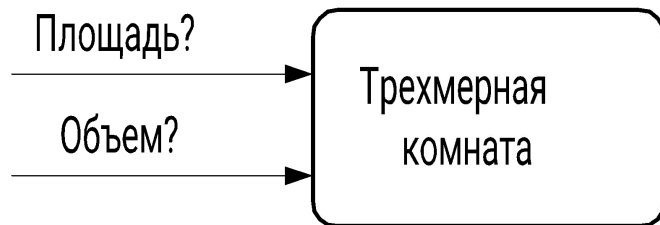
```
End.
```


Ограничение доступа (2)

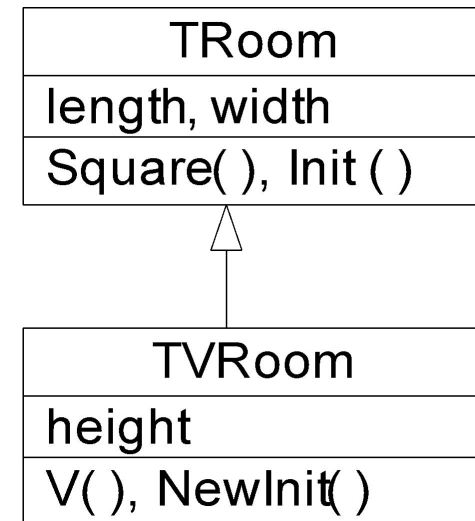
```
Program Ex_7_02;  
{ $APPTYPE CONSOLE }  
Uses SysUtils,  
      Room in 'Room.pas';  
Var A:TRoom;  
Begin  
  A.Init(3.5,5.1);  
  WriteLn('Room: length = ', A.length:6:2,  
          ' ; width = ', A.width:6:2);  
  WriteLn('Square = ', A.Square:8:2);  
  ReadLn;  
End.
```

7.3 Наследование

Наследование – механизм конструирования новых более сложных производных классов из уже имеющихся базовых посредством добавления полей и методов.



```
Program Ex_07_03;  
{ $APPTYPE CONSOLE }  
Uses SysUtils,  
      Room in Room.pas';  
Type TVRoom = object(TRoom)  
      height: single;  
      function V: single;  
      procedure NewInit(l, w, h: single);  
end;
```



Наследование (2)

```
Procedure TVRoom.NewInit;
```

```
Begin
```

```
    Init(l,w) ;
```

```
    height:=h;
```

```
End;
```

```
Function TVRoom.V;
```

```
Begin
```

```
    Result:=Square*height;
```

```
End;
```

```
Var A:TVRoom;
```

```
Begin
```

```
    A.NewInit(3.4,5.1,2.8) ;
```

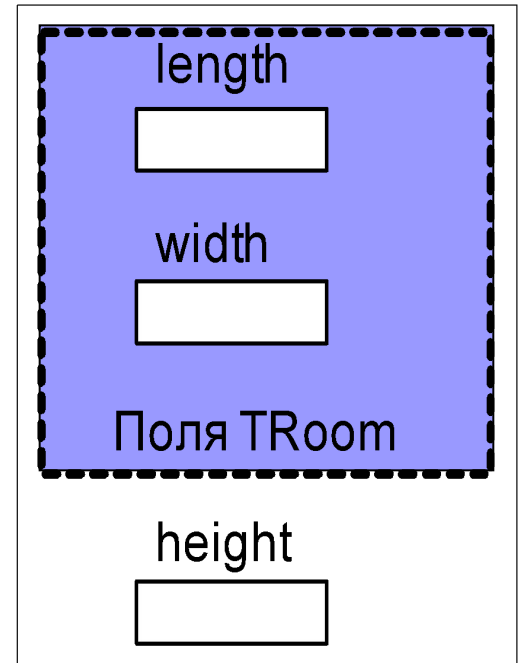
```
    WriteLn('Square = ', A.Square:6:2) ;
```

```
    WriteLn('V = ', A.V:6:2) ;
```

```
    ReadLn;
```

```
End.
```

Поля TVRoom



Присваивание объектов иерархии

Допустимо присваивать переменной типа базового класса значение переменной типа объекта производного класса.

```
Var A:TRoom;
```

```
    B:TVRoom;
```

```
...
```

```
A:=B; {допустимо}
```

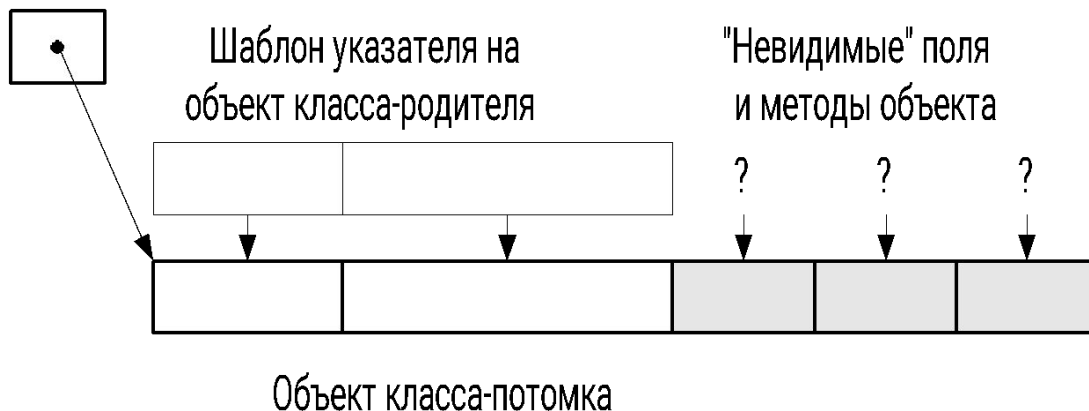
```
B:=A; {не допустимо!}
```

Присваивание указателей в иерархии

Допустимо указателю на объект базового класса присваивать адрес объекта производного класса.

Однако при этом возникает проблема «невидимых» полей.

Указатель на объект
класса-родителя



```
Var pC: ^TRoom;  
    E: TVRoom;
```

...

```
pC := @E;
```

```
S := pC^.Square;
```

```
V1 := pC^.V; {ошибка!}
```

```
Type pTVRoom = ^TVRoom;
```

```
Var pC: ^TRoom;
```

```
    E: TVRoom;
```

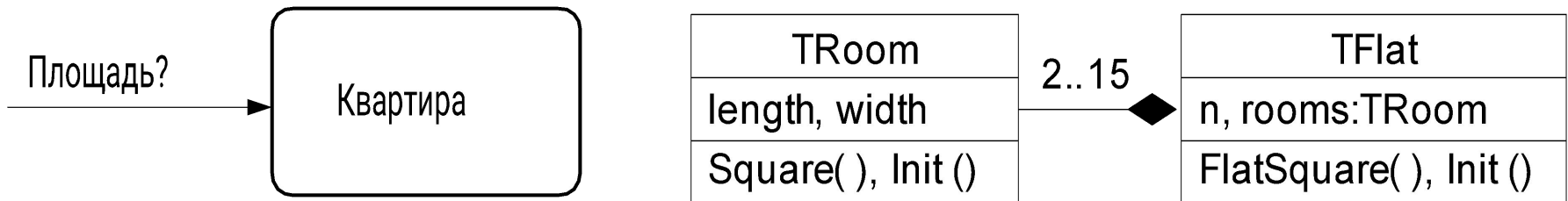
...

```
pC := @E;
```

```
V1 := pTVRoom(pC) ^.V;
```

7.4 КОМПОЗИЦИЯ

Композиция – включение объектов одного класса в объекты другого. Реализуется механизмом поддержки объектных полей.



```
Program Ex_7_04;
```

```
{$APPTYPE CONSOLE}
```

```
Uses SysUtils,
```

```
Room in 'Room.pas';
```

```
Type TFlat=object
```

```
    n:byte;
```

```
    rooms:array[1..15] of TRoom;
```

```
    function FlatSquare:single;
```

```
    procedure Init(an:byte;
```

```
        Const ar:array of TRoom);
```

```
end;
```

КОМПОЗИЦИЯ (2)

```
Procedure TFlat.Init;  
Var i:byte;  
Begin  
    n:=an;  
    for i:=1 to n do  
        rooms[i].Init(ar[i-1].length, ar[i-1].width);  
End;  
Function TFlat.FlatSquare;  
Var S:single; i:integer;  
Begin  
    S:=0;  
    for i:=1 to n do S:=S+rooms[i].Square;  
    Result:=S;  
End;
```

КОМПОЗИЦИЯ (3)

```
Var mas:array[1..3] of TRoom=  
    ((length:2.5; width:3.75),  
     (length:2.85; width:4.1),  
     (length:2.3; width:2.8));
```

```
Var F:TFlat;
```

```
Begin
```

```
    F.Init(3,mas);
```

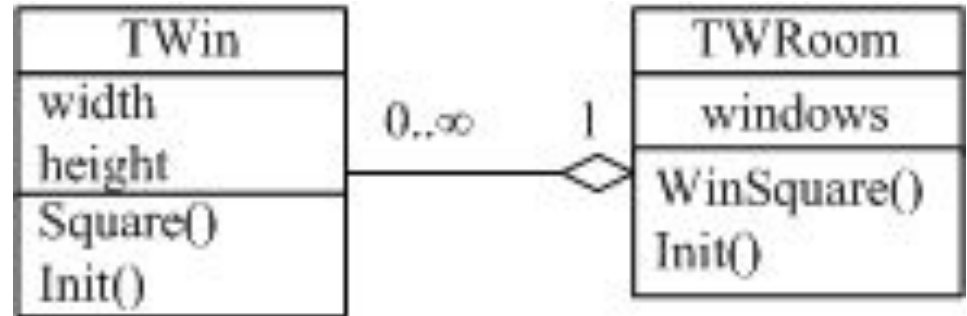
```
    WriteLn('S flat =',F.FlatSquare);
```

```
    ReadLn;
```

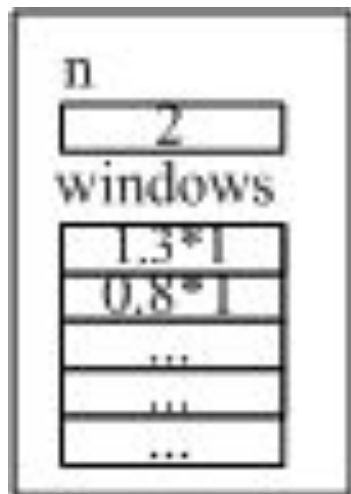
```
End.
```


7.5 Наполнение (агрегация)

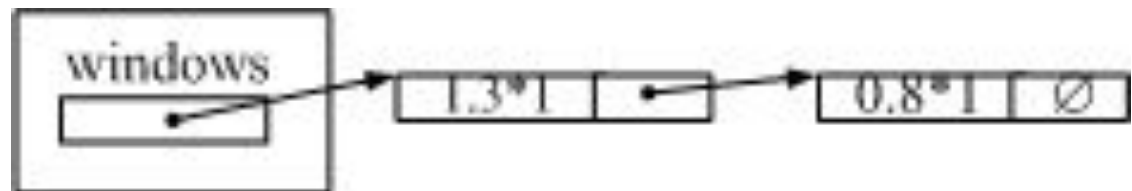
Наполнение – механизм построения классов, при котором объекты строящегося класса **могут** включать неопределенное количество: от 0 до сравнительно больших значений (на практике обычно до нескольких десятков), объектов других классов.



Реализация в виде массива окон



Реализация в виде списка окон



Наполнение (2)

```
Program Ex_7_05;
```

```
{$APPTYPE CONSOLE}
```

```
Uses SysUtils;
```

```
Type TWin = object {класс Окно}
```

```
    width,height:single;
```

```
    Function Square:single;
```

```
    Procedure Init(w,h:single);
```

```
end;
```

```
Function TWin.Square;
```

```
    Begin Result:= height*width; End;
```

```
Procedure TWin.Init;
```

```
    Begin width:=w; height:=h; End;
```

Наполнение (3)

```
Type pW = ^SWin;  
    SWin = record    // тип элемента списка  
        win:TWin;  
        p:pW;  
    end;  
  
Type TWRoom = object  
    windows:pW;  
    Function WinSquare:single;  
    Procedure Init(n:byte;Const ww:array of TWin);  
    Procedure Done; // освобождение памяти  
end;
```

Наполнение (4)

```
Procedure TWRoom.Init(n:byte;Const ww:array of TWin);
Var i:byte; q:pW;
Begin
  windows:=nil;
  if n<>0 then
    for i := 1 to n do
      begin
        new(q);
        // q^.win:=ww[i-1];
        q^.win.Init(ww[i-1].width,ww[i-1].height);
        if windows=nil then
          begin windows:=q; q^.p:=nil; end
          else
            begin q^.p:=windows; windows:=q; end;
      end;
    end;
  End;
```

Наполнение (5)

// процедура Done освобождает выделенную память

```
Procedure Done;
```

```
Var q:pW;
```

```
Begin
```

```
  while windows <> nil do
```

```
    begin
```

```
      q:= windows;
```

```
      windows := q^.p;
```

```
      dispose(q);
```

```
    end;
```

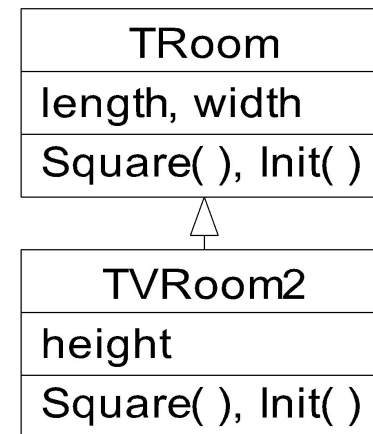
```
End;
```

Наполнение (6)

```
Function TWRoom.WinSquare;  
  Var q:pW;  
  Begin  
    Result:= 0;  q:=windows;  
    if q<>nil then  
      while q<>nil do  
        begin  
          Result:=Result+ q^.win.Square;  
          q:=q^.p;  
        end;  
      end;  
    End;  
  Var B:TWRoom;  
    W:array[1..3] of TWin = ((width:1.2; height:0.9),  
                             (width:1.8; height:0.9),  
                             (width:1.5; height:0.9));  
  Begin  
    B.Init(3,W);  
    WriteLn('Square =',B.WinSquare:8:2);  
    B.Done;  ReadLn;  
  End.
```

7.6 Простой полиморфизм

Простой (статический) полиморфизм – механизм переопределения методов при наследовании, при котором связь метода с объектом выполняется на этапе компиляции (*раннее связывание*).



```
Program Ex_7_06;
{$APPTYPE CONSOLE}
Uses SysUtils,
    Room in 'Ex_08_02\Room.pas';
Type TVRoom2 = object(TRoom)
    height:single;
    function Square:single;
    procedure Init(l,w,h:single);
end;
```

Простой полиморфизм (2)

```
Procedure TVRoom2.Init;  
  Begin  
    inherited Init(l,w);    { TRoom.Init(l,w); }  
    height:=h;  
  End;  
Function TVRoom2.Square;  
  Begin  
    Result:=2*(inherited Square+height*  
                                                       (length+width));  
  End;  
Var A:TVRoom2;  
Begin  
  A.Init(3.4,5.1,2.8);  
  WriteLn('Square = ',A.Square:6:2);  
  ReadLn;  
End.
```

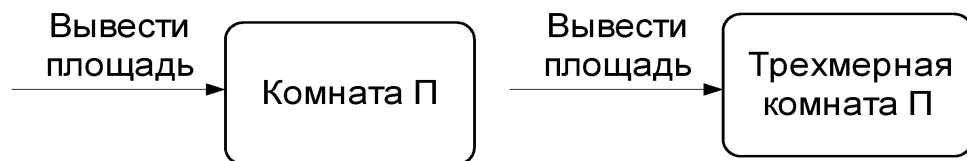

Обращение объекта производного класса к переопределенному методу базового класса в программе

При необходимости обращении к переопределенному методу базового класса явно меняют тип переменной – объекта класса, например так

```
Var A:TVRoom2;  
    B:TRoom;  
...  
    B:=A;  
    B.Square;
```

7.7 Сложный (динамический) полиморфизм. Конструкторы

Существует три ситуации, в которых определение *типа* объекта на этапе компиляции программы невозможно, и, следовательно, невозможно правильное подключение переопределенного метода.



```
Program Ex_7_07;
```

```
{$APPTYPE CONSOLE}
```

```
Uses SysUtils;
```

```
Type TRoomP=object
```

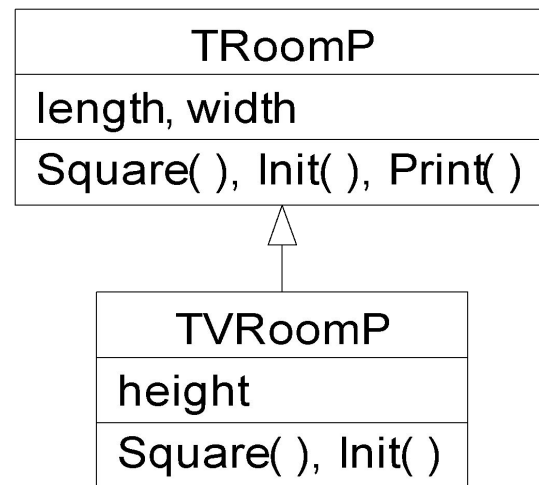
```
    length, width:single;
```

```
    function Square:single;
```

```
    procedure Print;
```

```
    procedure Init(l,w:single);
```

```
end;
```



СЛОЖНЫЙ ПОЛИМОРФИЗМ (2)

```
Function TRoomP.Square;  
    Begin Result:= length* width; End;  
Procedure TRoomP.Print;  
    Begin WriteLn('Square =', Square:6:2); End;  
Procedure TRoomP.Init;  
    Begin length:=1; width:=w; End;  
Type TVRoomP = object(TRoomP)  
    height:single;  
    function Square:single;  
    procedure Init(l,w,h:single);  
end;  
Procedure TVRoomP.Init;  
    Begin  
        inherited Init(l,w);  
        height:=h;  
    End;
```

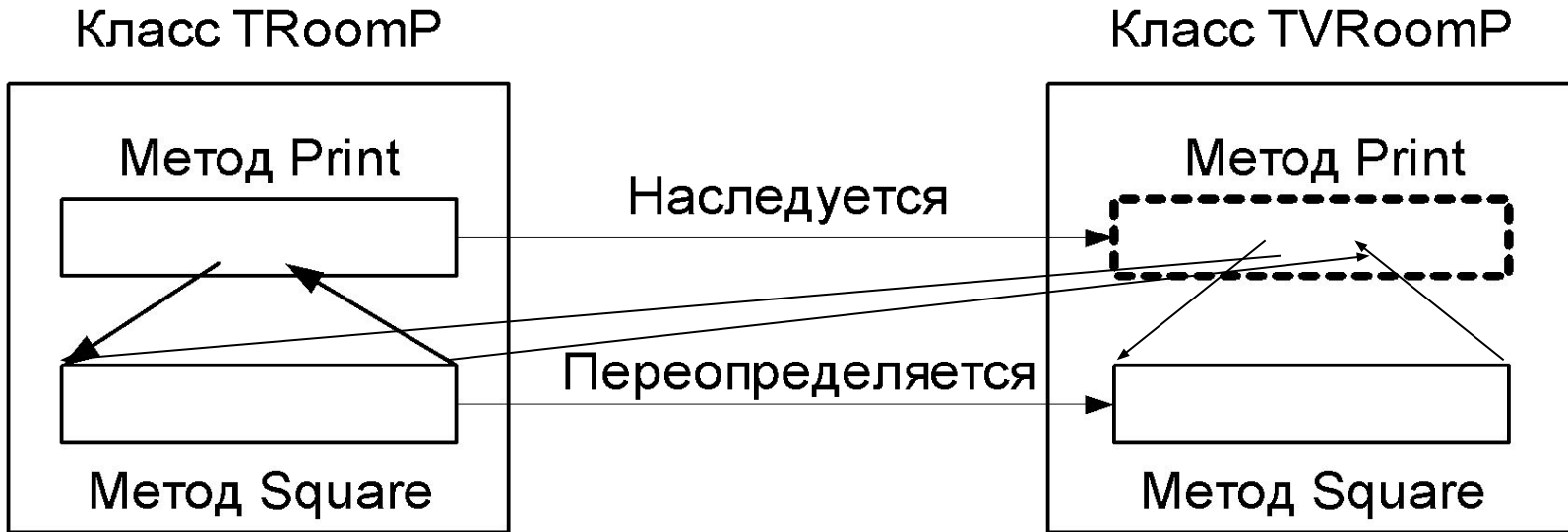
Сложный полиморфизм (2)

```
Function TVRoomP.Square;  
  Begin  
    Square:=2*(inherited Square+height*(length+width));  
  End;  
Var A:TRoomP; B:TVRoomP;  
Begin  
  A.Init(3.5,5.1);  
  A.Print;  
  B.Init(3.5,5.1,2.7);  
  B.Print;  
  ReadLn;  
End.
```

Square = 17.85
Square = 17.85

Ошибка!

Пояснение к ошибке



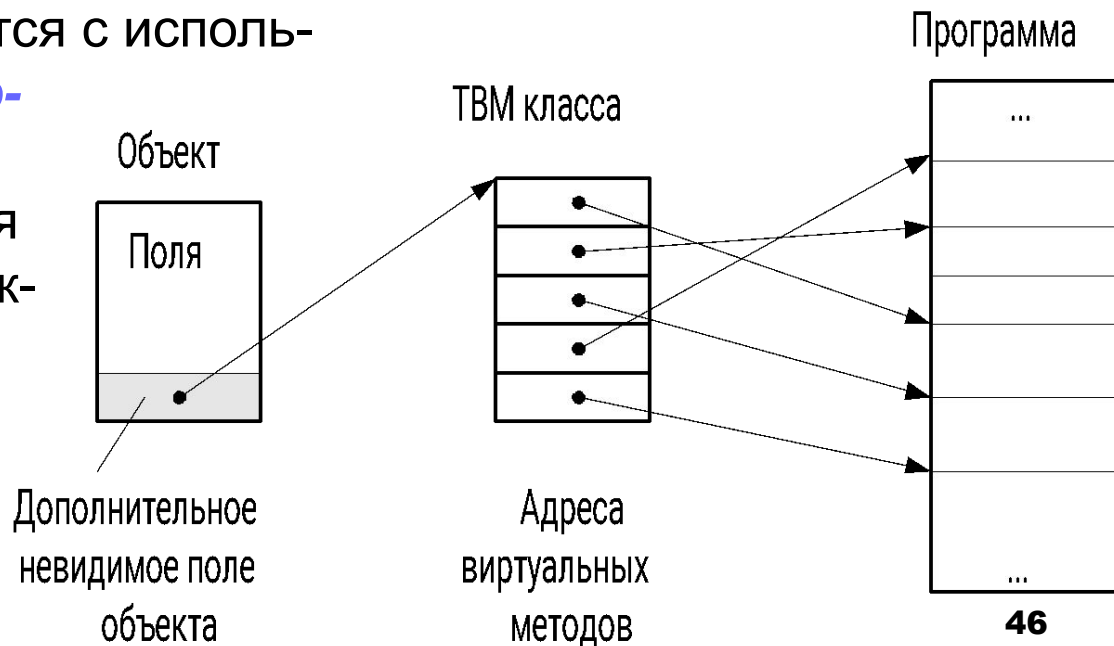
При **позднем связывании** нужный аспект полиморфного метода определяется на этапе выполнения программы по типу объекта, для которого вызывается метод.

Реализация сложного полиморфизма

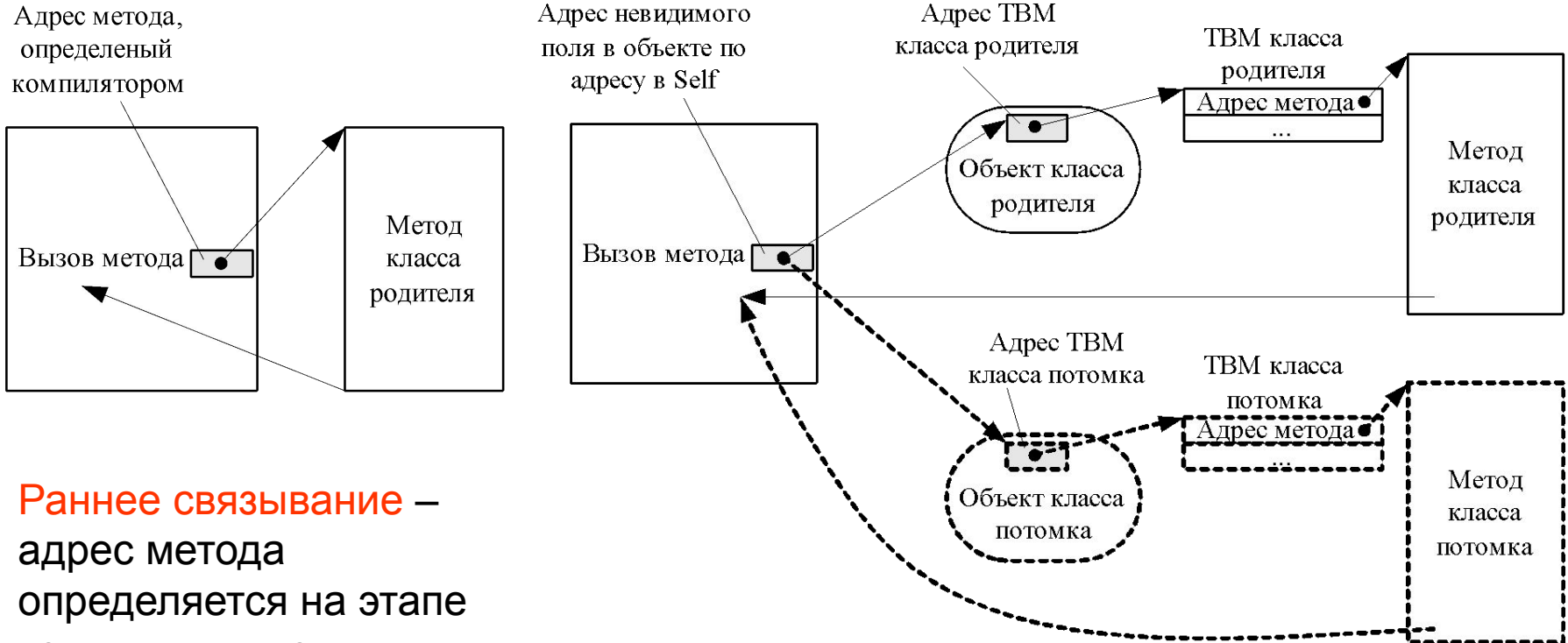
Для организации сложного полиморфизма необходимо:

- 1) переопределяемые методы описать служебным словом *virtual*;
- 2) к методам класса с виртуальными полиморфными методами добавить специальный метод-процедуру – *конструктор*, в котором служебное слово `procedure` заменено служебным словом `constructor`;
- 3) вызвать конструктор прежде, чем произойдет первое обращение к виртуальным полиморфным методам.

Подключение осуществляется с использованием *таблицы виртуальных методов* (ТВМ), которая создается при выполнении конструктора.



Различие раннего и позднего связывания



Раннее связывание – адрес метода определяется на этапе компиляции по *объявленному* типу переменной.

Позднее связывание – адрес метода определяется на этапе выполнения по *фактическому* типу объекта через таблицу виртуальных методов класса, адрес которой хранится в объекте.

Исправленный пример

```
Unit RoomP;
```

```
interface
```

```
Type TRoomP=object
```

```
    length, width:single;
```

```
    function Square:single; virtual;
```

```
    procedure Print;
```

```
    constructor Init(l,w:single);
```

```
end;
```

```
Type TVRoomP = object(TRoomP)
```

```
    height:single;
```

```
    function Square:single; virtual;
```

```
    constructor Init(l,w,h:single);
```

```
end;
```


Исправленный пример (2)

implementation

```
Function TRoomP.Square;
```

```
    Begin Result:= length* width; End;
```

```
Procedure TRoomP.Print;
```

```
    Begin WriteLn('Square =', Square:6:2); End;
```

```
Constructor TRoomP.Init;
```

```
    Begin length:=l; width:=w; End;
```

```
Constructor TVRoomP.Init;
```

```
    Begin
```

```
        inherited Init(l,w);
```

```
        height:=h;
```

```
    End;
```

```
Function TVRoomP.Square;
```

```
    Begin
```

```
        Square:=2*(inherited Square+height*(length+ width));
```

```
    End;
```

```
end.
```

Исправленный пример (3)

```
Program Ex_7_07a;  
{ $APPTYPE CONSOLE }  
Uses SysUtils,  
      RoomP in 'RoomP.pas';  
Var A:TRoomP; B:TVRoomP;  
Begin  
    A.Init(3.5,5.1);  
    A.Print;  
    B.Init(3.5,5.1,2.7);  
    B.Print;  
    ReadLn;  
End.
```

Square = 17.85

Square = 82.14

3 случая обязательного использования сложного полиморфизма

- 1-й случай** – если наследуемый метод для объекта производного класса вызывает метод, переопределенный в производном классе.
- 2-й случай** – если объект производного класса через указатель базового класса обращается к методу, переопределенному производным классом.
- 3-й случай** – если процедура вызывает переопределенный метод для объекта производного класса, переданного в процедуру через *параметр-переменную*, описанный как объект базового класса («процедура с полиморфным объектом»).

2-й случай

```
Program Ex_7_07b;  
{ $APPTYPE CONSOLE }  
Uses SysUtils,  
    RoomP in 'Ex_07_07\RoomP.pas';  
  
Var pA: ^TRoomP; B: TVRoomP;  
Begin  
    B.Init(3.5, 5.1, 2.7);  
    WriteLn('Square =', B.Square:6:2);  
    pA := @B;  
    WriteLn('Square =', pA^.Square:6:2);  
    ReadLn;  
end.
```

Square = 82.14

Square = 82.14

3-й случай. Процедура с полиморфным объектом

```
Program Ex_7_07c;  
{ $APPTYPE CONSOLE }  
Uses SysUtils,  
    RoomP in 'Ex_08_07\RoomP.pas';  
Procedure Print (Var R:TRoomP);  
Begin  
    WriteLn ('Square =', R.Square:6:2);  
End;  
Var A:TRoomP; B:TVRoomP;  
Begin  
    A.Init (3.5, 5.1);  
    B.Init (3.5, 5.1, 2.7);  
    Print (A);  
    Print (B);  
    ReadLn;  
End.
```

Square = 17.85

Square = 82.14

Функция определения типа полиморфного объекта

`TypeOf(<Имя класса или объекта>):pointer` – возвращает адрес ТВМ класса. Если адрес ТВМ объекта и класса совпадают, то объект является переменной данного класса.

Пример:

```
if TypeOf(Self) = TypeOf(<Имя класса>)
    then <Объект принадлежит классу>
    else <Объект не принадлежит классу>
```

Свойства виртуальных методов класса

- 1) позднее связывание требует построения ТВМ, а следовательно *больше памяти*;
- 2) вызов виртуальных полиморфных методов происходит через ТВМ, а следовательно *медленнее*;
- 3) *список параметров* одноименных виртуальных полиморфных методов *должен совпадать*, а статических полиморфных – не обязательно;
- 4) статический полиморфный метод не может переопределить виртуальный полиморфный метод.

7.8 Динамические полиморфные объекты. Деструкторы. Динамические поля

Создание полиморфных объектов:

Функция `New(<Тип указателя> [, <Вызов конструктора>])` – возвращает адрес размещенного и, при вызове конструктора, сконструированного объекта. Если вызов конструктора в `New` отсутствует, то после выделения памяти необходим вызов конструктора.

Деструктор – метод класса, который используется для корректного уничтожения полиморфного объекта, содержащего невидимое поле. Деструктор можно использовать для освобождения памяти динамических полей и переопределять при наследовании.

Уничтожение полиморфных объектов:

Процедура `Dispose (<Указатель> [, <Вызов деструктора>])` – если есть вызов деструктора, то устанавливается размер полиморфного объекта и корректно освобождается память.

Динамические полиморфные объекты (2)

```
Program Ex_7_08;  
{$APPTYPE CONSOLE}  
Uses SysUtils;  
Type pTRoomD = ^TRoomD;  
    TRoomD = object  
        length, width:single;  
        function Square:single; virtual;  
        constructor Init(l,w:single);  
        destructor Done;  
    end;  
Type pTVRoomD = ^TVRoomD;  
    TVRoomD = object(TRoomD)  
        height:single;  
        function Square:single; virtual;  
        constructor Init(l,w,h:single);  
    end;
```

Динамические полиморфные объекты (3)

```
Function TRoomD.Square;  
    Begin Result:= length* width; End;  
Constructor TRoomD.Init;  
    Begin length:=l; width:=w; End;  
Destructor TRoomD.Done;  
    Begin End;  
Constructor TVRoomD.Init;  
    Begin  
        inherited Init(l,w);  
        height:=h;  
    End;  
Function TVRoomD.Square;  
    Begin  
        Result:=2*(inherited Square+height*(length+ width));  
    End;
```

Динамические полиморфные объекты (4)

```
Var pA: pTRoomD; pB:pTVRoomD;
```

```
Begin
```

```
{указатель базового типа, объект базового типа}
```

```
pA:=New (pTRoomD, Init (3.5, 5.1));
```

```
WriteLn ('Square =', pA^.Square:6:2);
```

```
Dispose (pA, Done);
```

```
{указатель производного типа, объект производного типа}
```

```
pB:=New (pTVRoomD, Init (3.5, 5.1, 2.7));
```

```
WriteLn ('Square =', pB^.Square:6:2);
```

```
Dispose (pB, Done);
```

```
{указатель базового типа, объект производного типа}
```

```
pA:=New (pTVRoomD, Init (3.5, 5.1, 2.7));
```

```
WriteLn ('Square =', pA^.Square:6:2) Square =
```

```
Dispose (pA, Done);
```

```
ReadLn;
```

```
End.
```

17.85

Square =

82.14

Square =

Динамические поля в объектах

```
Program Ex_7_09;  
{$APPTYPE CONSOLE}  
Uses SysUtils;  
Type pTRoomD=^TRoomD;  
    TRoomD=object  
        length, width:single;  
        function Square:single; virtual;  
        constructor Init(l,w:single);  
        destructor Done; virtual;  
    end;  
Type pTBRoomD=^TBRoomD;  
    TBRoomD=object (TRoomD)  
        pB:pTRoomD;  
        function Square:single; virtual;  
        function BSquare:single;  
        constructor Init(l,w:single;  
                                                                    lb,wb:single);  
        destructor Done; virtual;  
    end;
```

Динамические поля в объектах (2)

```
Function TRoomD.Square;  
    Begin Square:= length* width; End;  
Constructor TRoomD.Init;  
    Begin length:=l; width:=w; End;  
Destructor TRoomD.Done;  
    Begin End;  
Constructor TBRoomD.Init;  
    Begin inherited Init(l,w);  
        if (lb=0) or (wb=0) then pB:=nil  
            else pB:= New(pTRoomD, Init(lb,wb));  
    End;  
Function TBRoomD.BSquare;  
    Begin if pB<>nil then BSquare:=pB^.Square  
        else BSquare:=0;  
    End;  
Function TBRoomD.Square;  
    Begin Square:= inherited Square+BSquare; End;  
Destructor TBRoomD.Done;  
    Begin if pB<>nil then Dispose(pB,Done); End;
```

Динамические поля в объектах (3)

```
Var A:TBRoomD; pB1:pTBRoomD; pB2:pTRoomD;
```

```
Begin
```

```
  {статический объект с динамическим полем}
```

```
  A.Init(3.2,5.1,2.5,1);
```

```
  WriteLn(A.Square:6:2,A.BSquare:6:2);
```

```
  A.Done;
```

```
  {динамический полиморфный объект с динамическим полем}
```

```
  pB1:=New(pTBRoomD,Init(3.2,5.1,2.5,1));
```

```
  WriteLn(pB1^.Square:6:2,pB1^.BSquare:6:2);
```

```
  Dispose(pB1,Done);
```

```
  {динамический полиморфный объект с динамическим полем}
```

```
  pB2:=new(pTBRoomD,Init(3.2,5.1,2.5,1));
```

```
  WriteLn(pB2^.Square:6:2,pTBRoomD(pB2)^.BSquare:6:2);
```

```
  Dispose(pB2,Done);
```

```
  ReadLn;
```

```
End.
```

18.82	2.50
18.82	2.50
18.82	2.50

