

Наследование и иерархии классов

Наумов Д.А., доц. каф. КТ

Основы программной инженерии, 2019

Содержание лекции

1 Класс как элемент абстракции

- Неявный параметр `self`
- Перегрузка методов
- Перегрузка арифметических операций

2 Наследование и иерархии классов

- Наследование как основная форма отношения обобщения
- Переопределение методов в производном классе
- Управление доступом к членам классам в связи с наследованием

Неявный параметр self

Для доступа к объекту класса, для которого вызван метод, в языке Pascal существует идентификатор **self**, обозначающее неявно передаваемый в метод объект.

```
type
  TComplex = class
    private
      _Re, _Im: extended;
    public
      constructor Create(Re, Im: extended);

      function GetRe: extended;
      function GetIm: extended;

      function Add(const Other: TComplex): TComplex;
    end;
```

При использовании идентификатора **self** метод выглядит следующим образом:

```
function TComplex.Add(const Other: TComplex): TComplex;
begin
  _Re := _Re + Other.GetRe;
  _Im := _Im + Other.GetIm;

  Result := self;
end;
```

Неявный параметр self

Пример использования метода Add:

```
var
  a, b, c: TComplex;

begin
  a := TComplex.Create(2, -3);
  b := TComplex.Create(4, 3);
  c := TComplex.Create(1, -1);

  //a.Add(b);
  //a.Add(c);

  a.Add(b).Add(c);

  writeln('re:', a.GetRe:0:4, ' im:', a.GetIm:0:4);

  readln;
end.
```

Перегрузка методов

Перегруженные методы используются, когда функции выполняют схожие по смыслу действия задачи с данными различных типов.

```
type
  TComplex = class
  private
    _Re, _Im: extended;
  public
    constructor Create(Re, Im: extended);

    function GetRe: extended;
    function GetIm: extended;

    function Add(const Other: TComplex): TComplex;
    function Add(const Other: extended): TComplex;
  end;
```

Реализация метода:

```
function TComplex.Add(const Other: extended): TComplex;
begin
  _Re := _Re + Other;

  Result := self;
end;
```

Обращение к методу:

```
a.Add(b).Add(c).Add(1.78);
```

Перегрузка арифметических операций

Перегрузка арифметических операций полезна тогда, когда хочется обеспечить привычную форму записи выражений при манипуляции с данными пользовательских типов.

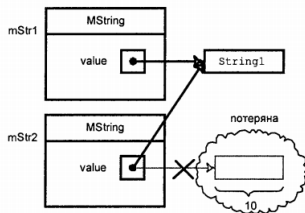
```
operator + (x, y: TComplex) r: TComplex;  
begin  
  r := TComplex.Create(x.GetRe + y.GetRe, x.GetIm * y.GetIm);  
end;
```

Запись выражений, оперирующих данными пользовательских типов в форме, привычной для выражений, оперирующих со встроенными типами, может в ряде случаев обеспечивать более наглядный и естественный код.

```
//a.Add(b).Add(c);  
a := a + b + c;
```

Поверхностное копирование объектов

В результате присваивания значение указателя value объекта mstr2 будет указывать на ту же память, что и указатель value объекта mstr1:

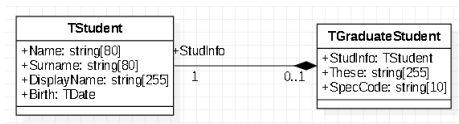


Пример:

```
writeln('re:', a.GetRe:0:4, ' im:', a.GetIm:0:4);  
writeln('re:', b.GetRe:0:4, ' im:', b.GetIm:0:4);  
  
a := b; |  
b.SetIm(0);  
  
writeln('re:', a.GetRe:0:4, ' im:', a.GetIm:0:4);  
writeln('re:', b.GetRe:0:4, ' im:', b.GetIm:0:4);
```

Использование агрегации

Выстраивание иерархии классов на основе отношения агрегации:



Код:

```

type
  //дата
  TDate = record Day, Month, Year: integer; end;

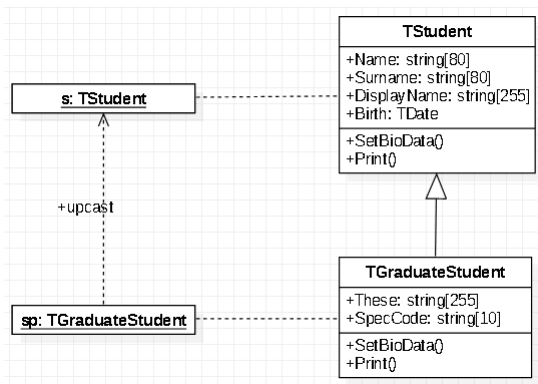
  //сведения о студенте
  TStudent = class
    Name,                               //имя
    Surname: string[80];                 //фамилия
    DisplayName: string[255];            //имя для печати
    Birth: TDate;                        //дата рождения
  end;

  //сведения об аспиранте
  TGraduateStudent = class
    StudentInfo: TStudent;               //аспирант как студент
    //только для аспиранта
    These: string[255];                  //тема диссертации
    SpecCode: string[10];                //код специальности
  end;
  
```

Недостатки: тип TGraduateStudent является независимым, его нельзя трактовать как разновидность типа TStudent.

Использование наследования

Объект типа TGraduateStudent является разновидностью объекта TStudent:



Использование наследования

```
type
  //дата
  TDate = record Day, Month, Year: integer; end;

  //сведения о студенте
  TStudent = class
  public
    Name,                               //имя
    Surname: string[80];                //фамилия
    DisplayName: string[255];           //имя для печати
    Birth: TDate;                       //дата рождения

    procedure SetBioData(AName, ASurname: string; ADay, AMonth, AYear: integer);

    procedure Print();
  end;

  //сведения об аспиранте
  TGraduantStudent = class(TStudent)
  public
    //только для аспиранта
    These: string[255];                 //тема диссертации
    SpecCode: string[10];               //код специальности

    procedure SetBioData(
      AName, ASurname: string;
      ADay, AMonth, AYear: integer;
      AThese: string;
      ASpecCode: string);

    procedure Print();
  end;
```

Использование наследования

```
procedure TStudent.SetBioData(AName, ASurname: string; ADay, AMonth,
    AYear: integer);
begin
    Name := AName;
    Surname := ASurname;
    DisplayName := AName + ' ' + ASurname;
    Birth.Day := ADay;
    Birth.Month := AMonth;
    Birth.Year := AYear;
end;

procedure TStudent.Print();
begin
    writeln('From TStudent.Print => Name: ', DisplayName);
end;
```

Использование наследования

```
procedure TGraduantStudent.SetBioData(  
  AName, ASurname: string;  
  ADay, AMonth, AYear: integer;  
  AThese: string;  
  ASpecCode: string);  
begin  
  inherited SetBioData(AName, ASurname, ADay, AMonth, AYear);  
  
  These := AThese;  
  SpecCode := ASpecCode;  
end;
```

```
procedure TGraduantStudent.Print();  
begin  
  inherited Print();  
  
  writeln('From TGraduateStudent.Print => These: ', These);  
end;
```

Использование наследования

```
procedure PrintBirthDay(const s: TStudent);  
begin  
    write('From PrintBirthDay() => ');  
    writeln(s.Birth.Day, '.', s.Birth.Month, '.', s.Birth.Year);  
end;
```

```
procedure InformGarduate(const gs: TGraduateStudent);  
begin  
    //получение gs типа TGraduateStudent  
    //...  
    write('From InformGarduate() => ');  
  
    //и передача его дальше как TStudent  
    PrintBirthDay(gs);  
end;
```

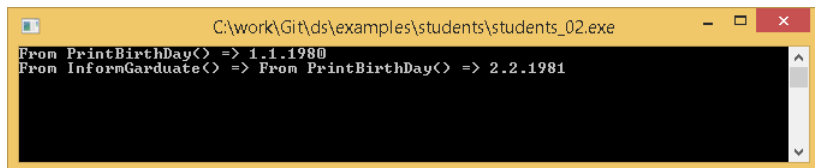
Использование наследования

```
var
  s: TStudent;
  gs: TGraduateStudent;

begin
  s := TStudent.Create;
  s.SetBioData('Ivan', 'Petrov', 1, 1, 1980);
  PrintBirthDay(s);
  s.Free;

  gs := TGraduateStudent.Create;
  gs.SetBioData('Oleg', 'Sidorov', 2, 2, 1981,
    'Space objects detection', '09.04.01');
  InformGarduate(gs);
  gs.Free;

  readln;
end.
```



Преимущества использования наследования

- возможность повторного использования кода (это достигается и агрегированием);
- возможность отразить взаимоотношения объектов, свойственные предметной области;
- возможность рассматривать производный класс как подкласс базового, упрощая реализацию кода для производного класса;
- возможность обработки производных классов методами, разработанными при проектировании базового класса;
- возможность обновления или изменения содержания этих методов применительно к объектам производного класса, если такая необходимость имеется.

Обращение к одноименным методам

```
procedure CallPrint (ps: PStudent);  
begin  
    ps^.Print;  
end;
```

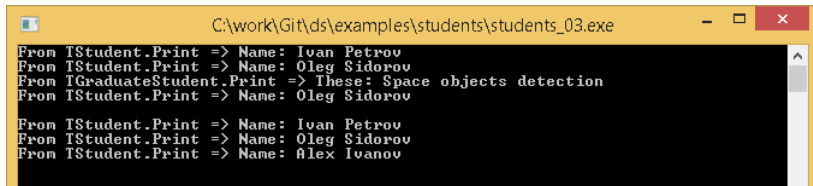
```
var  
    s: TStudent;  
    gs1, gs2: TGraduateStudent;  
  
    ps: ^TStudent;  
  
    parr: array[1..3] of ^TStudent;  
    i: integer;
```

Обращение к одноименным методам

```
begin
```

```
  s := TStudent.Create;  
  s.SetBioData('Ivan', 'Petrov', 1, 1, 1980);  
  
  gs1 := TGraduateStudent.Create;  
  gs1.SetBioData('Oleg', 'Sidorov', 2, 2, 1981,  
    'Space objects detection', '09.04.01');  
  
  gs2 := TGraduateStudent.Create;  
  gs2.SetBioData('Alex', 'Ivanov', 3, 3, 1982,  
    'Image classification', '09.04.02');  
  
  s.Print;           //вызывается процедура TStudent.Print  
  gs1.Print;         //вызывается процедура TGraduateStudent.Print  
  
  ps := @gs1;  
  ps^.Print();       //вызывается процедура TStudent.Print  
  
  writeln;  
  
  parr[1] := @s;  
  parr[2] := @gs1;  
  parr[3] := @gs2;  
  
  for i := 1 to 3 do  
    CallPrint(parr[i]); //вызывается процедура TStudent.Print  
  
  readln;
```

Обращение к одноименным методам



```
C:\work\Git\ds\examples\students\students_03.exe

From IStudent.Print => Name: Ivan Petrov
From IStudent.Print => Name: Oleg Sidorov
From IGraduateStudent.Print => These: Space objects detection
From IStudent.Print => Name: Oleg Sidorov

From IStudent.Print => Name: Ivan Petrov
From IStudent.Print => Name: Oleg Sidorov
From IStudent.Print => Name: Alex Ivanov
```

Переопределение методов в производном классе

Программист может переопределить унаследованную функцию-член базового класса, если содержание метода его не устраивает.

```
//сведения о студенте
TStudent = class
public
  //...
  //метод базового класса
  procedure Print(); //печатать информации о студенте
end;
```

```
//сведения об аспиранте
TGraduateStudent = class(TStudent)
public
  //...
  //переопределение метода в производном классе
  procedure Print(); //печатать информации об аспиранте
end;
```

При этом сохраняет возможность обратиться к методу базового класса:

```
procedure TGraduateStudent.Print();
begin
  //обращение к методу базового класса
  inherited Print();

  //...
end;
```

Инициализация объектов производного класса

Если базовый класс имеет конструктор, он должен быть вызван. Конструкторы по умолчанию могут быть вызваны неявно. Если базовый класс не имеет конструктора по умолчанию, следует обеспечить его явный вызов.

```
type
  //базовый класс
  TBase = class
    _x: integer;
  public
    constructor Create(x: integer); //конструктор базового класса
  end;

  //производный класс
  TDerived = class(TBase)
    _y: integer;
  public
    constructor Create(x, y: integer); //конструктор производного класса
  end;

constructor TBase.Create(x: integer); //конструктор базового класса
begin
  _x := x;

  writeln('Base initialized by value ', x);
end;

constructor TDerived.Create(x, y: integer); //конструктор производного класса
begin
  inherited Create(x);

  _y := y;

  writeln('Derived initialized by value ', y);
end;
```

Инициализация объектов производного класса

```
constructor TBase.Create(x: integer); //конструктор базового класса
begin
  _x := x;

  writeln('Base initialized by value ', x);
end;

constructor TDerived.Create(x, y: integer); //конструктор производного класса
begin
  inherited Create(x);


  _y := y;

  writeln('Derived initialized by value ', y);
end;

var
  d: TDerived;

begin
  d := TDerived.Create(5, 10);
  readln;
end.
```

В результате будет напечатано:



```
C:\work\Git\ds\examples\lec_06\ex_02.exe
Base initialized by value 5
Derived initialized by value 10
-
```

Порядок вызовов конструкторов и деструкторов

```
type
//базовый класс
TRoot = class
    constructor Create; //конструктор базового класса
    destructor Destroy; override; //деструктор
end;

//базовый класс
TBase = class(TRoot)
    constructor Create(); //конструктор производного класса
    destructor Destroy; override; //деструктор
end;

//производный класс
TDerived = class(TBase)
    constructor Create(); //конструктор производного класса
    destructor Destroy; override; //деструктор
end;

//производный класс
TMostDerived = class(TDerived)
    constructor Create(); //конструктор производного класса
    destructor Destroy; override; //деструктор
end;
```

Порядок вызовов конструкторов и деструкторов

```
constructor TRoot.Create();  
begin  
  writeln('Root constructor called');  
end;
```

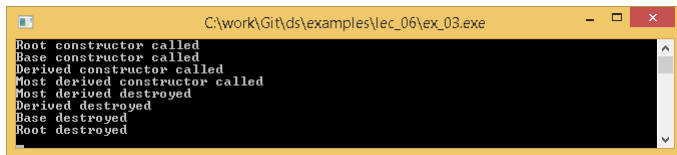
```
destructor TRoot.Destroy;  
begin  
  writeln('Root destroyed');  
  inherited Destroy;  
end;
```

```
constructor TBase.Create();  
begin  
  inherited Create;  
  writeln('Base constructor called');  
end;
```

```
destructor TBase.Destroy;  
begin  
  writeln('Base destroyed');  
  inherited Destroy;  
end;
```

Порядок вызовов конструкторов и деструкторов

```
var  
  d: TMostDerived;  
  
begin  
  d := TMostDerived.Create;  
  
  d.Free;  
  
  readln;  
end.
```



```
C:\work\Git\ds\examples\lec_06\ex_03.exe  
Root constructor called  
Base constructor called  
Derived constructor called  
Most derived constructor called  
Most derived destroyed  
Derived destroyed  
Base destroyed  
Root destroyed
```


Управление доступом к членам классам в связи с наследованием

- члены класса, объявленные со спецификатором `public`, доступны как в функциях-членах класса, так и за их пределами;
- члены класса, объявленные со спецификатором `private`, доступны только в функциях-членах класса;
- если необходимо предоставить доступ к членам класса для функций-членов производного класса, но при этом закрыть доступ извне, используется спецификатор доступа `protected`.

```
type
  TBase = class
  public
    procedure PublicMethod();
  protected
    procedure ProtectedMethod();
  private
    procedure PrivateMethod();
  end;
```

Управление доступом к членам классам в связи с наследованием

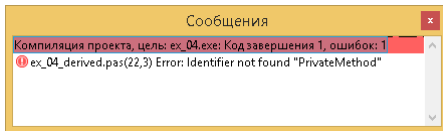
```
type
  TDerived = class(TBase)
  public
    procedure TestMethod();
  end;

implementation

procedure TDerived.TestMethod();
begin
  PublicMethod();

  ProtectedMethod();

  PrivateMethod();
end;
```



Однокоренная иерархия классов

Однокоренная иерархия

предполагает, что все классы неявно унаследованы (прямо или опосредованно) от суперкласса TObject.

Класс TObject определяет набор операций, общих для всех объектов.

```
type
  TMyClass = class

end;

var
  c: TMyClass;

begin
  c.
end. function      MethodName (address:codepo ^
function      newinstance :tobject
function      SafeCallException (exceptol
function      StringMessageTable :pstring
function      ToString :ansistring
function      UnitName :ansistring
```

Методы класса TObject могут замещаться в производных классах.