



UNIVERSIDADE SALGADO DE OLIVEIRA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO
TEORIA DA COMPUTAÇÃO E ALGORITMOS
Prof. Giuliano Prado de Moraes Giglio

DELPHI

Orientação a Objetos

+

Object Pascal

2º Semestre de 2003

Conceitos de Orientação a Objetos

A Metodologia Orientada a Objetos é relativamente nova, pois por volta de 1970 surgiram as primeiras publicações, mas o seu "boom" se deu nos anos 90, quando tornou-se a principal metodologia de desenvolvimento de software. A orientação a objetos permite modelar de forma mais natural o mundo real, pois as estruturas de dados são vistas como objetos, ou seja, têm características e funções. Seu maior objetivo é aumentar a produtividade do desenvolvimento de software através de uma maior expansibilidade e reutilização de código, além de controlar a complexidade e o custo da manutenção do mesmo.

Quando a metodologia orientada a objetos é utilizada, a fase de projeto do desenvolvimento do software está mais intimamente ligada à fase de implementação. Um dos pontos-chaves da metodologia orientada a objetos é centralização das atenções nas Estruturas de Dados, ao contrário da metodologia estruturada, onde a atenção era centralizada nos procedimentos. Na orientação a objetos há uma maior aproximação entre dados e procedimentos, pois procedimentos são definidos em termos dos dados.

Componentes Básicos da Orientação a Objetos

Classe	→ São moldes através dos quais criamos objetos
Objeto	→ Abstração que agrupa características e comportamentos.
Instância	→ É o objeto propriamente dito. Possui características próprias.
Propriedade	→ Define as características dos objetos de uma classe.
Método	→ Define o comportamento dos objetos de uma classe.
Mensagem	→ Representa uma ação do objeto ou uma mudança de estado. Define a comunicação entre objetos.
Interface	→ Conjunto de mensagens que define o comportamento de um objeto (Protocolo).

Encapsulamento

- É a capacidade de "esconder" detalhes de implementação (abstração).
- O principal objetivo é tornar o objeto independente de sua implementação interna, para isso a implementação das suas propriedades e métodos são "escondidas" de forma que o usuário precise apenas conhecer a interface do objeto para poder utilizá-lo. Desta forma o desenvolvedor poderá alterar tranquilamente a implementação de um objeto (Classe) sem causar transtornos aos usuários.

Herança

- Permite que uma nova classe seja descrita a partir de outra classe já existente (Reutilização).
- A *subclasse* herda as características e o comportamento da *superclasse*.
- A *subclasse* pode adicionar novas características e comportamentos aos herdados da *superclasse*.
- A *subclasse* pode ter um comportamento diferente da *superclasse*, redefinindo o método herdado.
- A *subclasse* é uma especialização da *superclasse*.
- Toda instância da *subclasse* é também uma instância da *superclasse*.
- O resultado de uma seqüência de heranças é uma *hierarquia de classes*.

Classes Abstratas

- Definida para ser a base de uma hierarquia.
- Possui métodos que não estão implementados.
- Não pode ser instanciada.

Polimorfismo

- É a capacidade de tratarmos objetos de diferentes tipos de uma mesma maneira desde que eles tenham um ancestral em comum
- Objetos de classes diferentes podem ter métodos com mesmo nome e cada objeto responderá adequadamente de acordo com seu método.
- Métodos da mesma classe podem ter o mesmo nome, desde que possuam quantidade ou tipo de parâmetros diferentes.
- Métodos da classe derivada podem ter nomes iguais aos da classe base, inclusive com parâmetros iguais.

Object Pascal

Estrutura de uma Unit

Unit <identificador>;

Interface

{Especifica o que será exportado pela UNIT afim de ser utilizados por outros módulos }

[uses <lista de units>;]

<seções de declaração>

Implementation

{Declaração de Variáveis, Constante e tipos locais a UNIT e Implementação dos métodos.

[uses <lista de units>;]

<definições>

[Initialization

{Código executado automaticamente quando um aplicativo que utiliza a UNIT é executado}

<instruções>]

[Finalization

{Código executado automaticamente quando um aplicativo que utiliza a UNIT é finalizado}

<instruções>]

end.

Classes

Definição de uma Classe

```
<identificador> = class [(Descendência)]
                  <atributos e métodos>
                  end;
```

- Deve ser feita na seção de declaração de tipos principal de um programa ou de uma unit.
- No caso do Delphi, todas as classes são descendentes de **TObject**.

Atributos e Métodos

```
<visibilidade 1>
  <lista de variáveis>
  <lista de procedimentos ou funções>
<visibilidade 2>
  <lista de variáveis>
  <lista de procedimentos ou funções>
  .
  .
  .
<visibilidade n>
  <lista de variáveis>
  <lista de procedimentos ou funções>
```

Visibilidade

- Define quem tem permissão de acessar e alterar os atributos e métodos da classe.
- Em uma mesma classe pode existir atributos e métodos com visibilidades diferentes.

Visibilidade	Descrição
Public	Os atributos e métodos podem ser manipulados por qualquer classe.
Private	Os atributos e métodos só podem ser manipulados pela própria classe.
Protected	Os atributos e métodos podem ser manipulados pela própria classe ou por qualquer subclasse desta e por demais classes declaradas na mesma UNIT.
Published	Semelhante a visibilidade public sendo que permite o acesso em tempo de projeto.

Declarando, Instanciando, Destruindo e Referenciando Objetos

- É declarado da mesma maneira que uma variável.
- Para que um objeto possa ser utilizado, este deve ser instanciado e após o seu uso o mesmo deve ser liberado
- Se comporta como um ponteiro, mas é manipulado como uma variável normal.
- Pode ser atribuído o valor **nil**.

```
<Objeto> : Classe;           {Declarando}

<Objeto> = Classe.Create;    {Instanciando}

<Objeto>.Free;              {Destruindo}

<Objeto>.Identificador     {Referenciando}
```

,onde Identificador representa uma propriedade ou um método. A referência a Dados é idêntica a referência a código.

Método Construtor

```
constructor <identificador> ( <parâmetros formais> );
```

- Aloca memória e inicializa o objeto, baseado nos parâmetros passados.
- Normalmente a primeira ação é invocar o construtor da classe base, através da instrução:

```
inherited <construtor> ( <parâmetros reais > );
```

Método Destrutor

```
destructor <identificador> ( <parâmetros formais> );
```

- Destroi o objeto, baseado nos parâmetros passados, e libera a memória alocada para ele.
- Normalmente a última ação é invocar o destrutor da classe base, através da instrução:

```
inherited <destrutor> ( <parâmetros reais > );
```

O Parâmetro Self

- Representa um parâmetro invisível passado a todos os métodos de uma classe e representa a instância da classe que esta chamando o método.
- É utilizado para evitar conflitos de nomes de objetos

Métodos Estáticos

- Os métodos declarados numa classe são por default estáticos
- Tem suas referências determinadas em tempo de compilação

Métodos Virtuais

- O objetivo dos métodos virtuais é a possibilidade de substituí-los por novos métodos, contendo os mesmo parâmetros, das classes descendentes.
- Para tornar um método virtual, basta acrescentar no final de sua declaração na classe, a palavra *virtual*;
- Um método virtual pode ser substituído em uma classe descendente através de sua redeclaração seguida da diretiva *override*;

Métodos Dinâmicos

- Métodos dinâmicos são basicamente idênticos a métodos virtuais sendo declarados com o uso da diretiva *dynamic*

OBS :Os métodos dinâmicos favorecem o tamanho do código enquanto os métodos virtuais favorecem a velocidade.

Métodos Abstratos

- São métodos que não fazem nada, servem apenas para definir a estrutura de uma hierarquia.
- Para tornar um método abstrato, basta acrescentar no final de sua declaração na classe, a palavra *abstract*;
- Para um método ser abstrato, é necessário que ele seja também virtual.

Propriedades

- Representa um mecanismo para encapsular os campos de uma Classe sobrecarregando as operações de leitura e escrita
- São uma extensão natural às variáveis de instância de uma classe, pois permitem que o desenvolvedor pareça estar trabalhando com estas, enquanto na realidade está executando chamadas a métodos.
- Para utilizar as propriedades os *campos* (atributos) devem ser declarados como **private**, os *métodos* como **protected**, e as *propriedades* como **public**.

```
property Identificador : TIPO  
    [read MétodoDeLeitura]  
    [write MétodoDeEscrita];
```

onde, Identificador representa a propriedade, TIPO o tipo da propriedade, MétodoDeLeitura o método associado a leitura da propriedade, MétodoDeEscrita o método associado a escrita da propriedade.

Verificação de Tipo

- Verifica, em tempo de execução, se o objeto é uma instância da classe ou de alguma subclasse desta.
- Retorna *true* ou *false*.

```
<objeto> is <classe>
```

Conversão de Tipo

- Converte, se possível, uma referência para o objeto de uma classe base em uma referência para objeto da subclasse.

```
(<objeto> as <classe>).<Métodos ou Propriedade>
```

Exemplos de Orientação a Objetos

```
unit Datas;
```

```
interface
```

```
type
```

```
  TData = Class(TObject)
    private
      Dia, Mes, Ano : Integer;
    public
      constructor Init (d,m,a : integer);
      procedure DefVal (d,m,a : integer);
      function  AnoBis : boolean;
      procedure Incrementa;
      procedure Decrementa;
      procedure Adiciona (NumDeDias : integer);
      procedure Subtrai (NumDeDias : integer);
      function  GetText : string;
    private
      function DiasNoMes : Integer;
  end;
```

```
implementation
```

```
constructor TData.Init (d,m,a : integer);
begin
  dia := d; Mes := m; ano := a;
end;
```

```
procedure TData.DefVal (d,m,a : integer);
begin
  dia := d; Mes := m; ano := a;
end;
```

```
function TData.AnoBis : boolean;
begin
  if (ano mod 4 <> 0)
  then AnoBis := false
  else
    if (ano mod 100 <> 0)
    then AnoBis := true
    else
      if (ano mod 400 <> 0)
      then AnoBis := False
      else AnoBis := True;
end;
```

```
function TData.DiasNoMes : integer;
begin
  case Mes of
    1,3,5,7,8,10,12 : DiasNoMes := 31;
    4,6,9,11       : DiasNoMes := 30;
    2 : if (AnoBis)
        then DiasNoMes := 29
        else DiasNoMes := 28;
  end;
end;
```

```
end;
```

```
procedure TData.Incrementa;
begin
  if (dia < DiasNoMes) {se não for o último dia do Mes}
  then inc(dia)
  else
    if (Mes < 12) {se não for dezembro}
    then
      begin
        inc(Mes);
        dia := 1;
      end
    else {se for o dia de ano novo}
    begin
      inc(ano);
      Mes := 1;
      dia := 1;
    end;
end;
```

```
procedure TData.Decrementa;
begin
  if (dia > 1)
  then Dec(dia) {se não for o primeiro dia do mês}
  else
    if (Mes > 1) {se não for o primeiro dia do ano}
    then
      begin
        Dec(Mes);
        dia := DiasNoMes;
      end
    else
      begin
        Dec(ano);
        Mes := 12;
        dia := DiasNoMes;
      end;
end;
```

```
function TData.GetText : string;
var d, m, a : string;
begin
  d := IntToStr(dia);
  case Mes of
    1 : m := 'Janeiro';
    2 : m := 'Fevereiro';
    3 : m := 'Março';
    4 : m := 'Abril';
    5 : m := 'Maio';
    6 : m := 'Junho';
    7 : m := 'Julho';
    8 : m := 'Agosto';
    9 : m := 'Setembro';
    10: m := 'Outubro';
    11: m := 'Novembro';
    12: m := 'Dezembro';
  end;
  a := IntToStr(ano);
  GetText := d + ', ' + m + ' de ' + a;
end;
```

```
procedure TData.Adiciona (NumDeDias : integer);
var n : integer;
begin
  for n := 1 to NumDeDias do
    Incrementa;
end;

procedure TData.Subtrai (NumDeDias : integer);
var n : integer;
begin
  for n := 1 to NumDeDias do
    Decrementa;
end;

end.
```

```
unit Universidade;

interface

type

TEndereco = Class
  private
    FRua, FBairro, FNumero : String;
  protected
    function  GetRua      : string;
    function  GetBairro  : string;
    function  GetNumero  : string;
    procedure SetRua     (Value : string);
    procedure SetBairro (Value : string);
    procedure SetNumero (Value : string);
  public
    property Rua      : string read GetRua      write SetRua;
    property Bairro  : string read GetBairro   write SetBairro;
    property Numero  : string read GetNumero   write SetNumero;
End;

TPessoa = Class
  private
    FMatricula : string;
    FNome      : string;
  protected
    function  GetMatricula : string;
    function  GetNome      : string;
    procedure SetMatricula (Value : string);
    procedure SetNome      (Value : string);
  public
    Endereco : TEndereco;
    constructor Create;
    destructor Destroy; override;
    property Matricula : string read GetMatricula write SetMatricula;
    property Nome      : string read GetNome      write SetNome;
End;

TProfessor = Class(TPessoa)
  private
    FDepartamento : string;
    FAulasMes      : byte;
  protected
    function  GetDepartamento : string;
    function  GetAulasMes      : byte;
    procedure SetDepartamento (Value : string);
    procedure SetAulasMes      (Value : byte);
  public
    property Departamento : String read GetDepartamento write SetDepartamento;
    property AulasMes     : byte   read GetAulasMes       write SetAulasMes;
End;
```

```
{
  TAluno = Class(TPessoa)
  private
    FCurso : string;
    FMGP    : real;
  protected
    function GetCurso : string;
    function GetMGP    : real;
    procedure SetCurso (Value : string);
    procedure SetMGP    (Value : byte);
  public
    property Curso : string read GetCurso write SetCurso;
    property MGP    : string read GetMGP    write SetMGP;
End;

TFuncionario = Class(TPessoa)
  private
    FSetor      : string;
    FSalario    : real;
  protected
    function GetSetor      : string;
    function GetSalario    : real;
    procedure SetSetor    (Value : string);
    procedure SetSalario  (Value : byte);
  public
    property Setor      : string read GetSetor    write SetSetor;
    property Salario : real   read GetSalario  write SetSalario;

  End;
}
implementation

{***** TEndereco*****}

function TEndereco.GetRua : string;
Begin
  Result := Self.FRua;
End;

function TEndereco.GetBairro : string;
Begin
  Result := Self.FBairro;
End;

function TEndereco.GetNumero : string;
Begin
  Result := Self.FNumero;
End;

procedure TEndereco.SetRua (Value : string);
Begin
  Self.FRua := Value;
End;

procedure TEndereco.SetBairro (Value : string);
Begin
  Self.FBairro := Value;
End;

procedure TEndereco.SetNumero (Value : string);
Begin
  Self.FNumero := Value;
End;
```

```
{***** TPessoa*****}  
  
constructor TPessoa.Create;  
Begin  
    inherited Create;  
    Self.Endereco := TEndereco.Create;  
End;  
  
destructor TPessoa.Destroy;  
Begin  
    Self.Endereco.Free;  
    inherited Destroy;  
End;  
  
function    TPessoa.GetMatricula : string;  
Begin  
    Result := Self.FMatricula;  
End;  
  
function    TPessoa.GetNome      : string;  
Begin  
    Result := Self.FNome;  
End;  
  
procedure   TPessoa.SetMatricula (Value : string);  
Begin  
    Self.FMatricula := Value;  
End;  
  
procedure   TPessoa.SetNome      (Value : string);  
Begin  
    Self.FNome := Value;  
End;  
  
{***** TProfessor*****}  
  
function TProfessor.GetDepartamento : string;  
Begin  
    Result := Self.FDepartamento;  
End;  
  
function TProfessor.GetAulasMes : byte;  
Begin  
    Result := Self.FAulasMes;  
End;  
  
procedure Tprofessor.SetDepartamento (Value : string);  
Begin  
    Self.FDepartamento := Value;  
End;  
  
procedure Tprofessor.SetAulasMes (Value : byte);  
Begin  
    Self.FAulasMes := Value;  
End;  
  
end.
```

```
unit Figura;

interface

type
  TFigura = Class
    private
      FCor      : string;
    protected
      function  GetCor   : string;
      procedure SetCor (Value : string);
    public
      procedure Desenhe; virtual; abstract;
      property  Cor : string read GetCor write SetCor;
End;

  TQuadrado = Class (TFigura)
    private
      FX : real;
    protected
      function  GetX : real;
      procedure SetX (Value : real);
    public
      procedure Desenhe ; override;
      property X : real read  GetX write SetX;
End;

  TRetangulo = Class (TQuadrado)
    private
      FY : real;
    protected
      function  GetY : real;
      procedure SetY (Value : real);
    public
      procedure Desenhe; override;
      property Y : real read GetY write  SetY;
End;

  TCirculo = Class (TFigura)
    private
      FRaio : real;
    protected
      function  GetRaio : real;
      procedure SetRaio (Value : Real);
    public
      procedure Desenhe ;override;
      property Raio : real read GetRaio write SetRaio;
End;

implementation

uses dialogs, SysUtils;

{*****TFigura*****}

function  TFigura.GetCor   : string;
Begin
  Result := Self.FCor;
End;

procedure TFigura.SetCor (Value : string);
Begin
  Self.FCor := Value;
End;
```

```
{*****TQuadrado*****}  
  
procedure TQuadrado.Desenhe;  
Begin  
  showMessage('Quadrado' + FloatToStr(Self.X));  
End;  
  
function TQuadrado.GetX : real;  
Begin  
  Result := Self.FX;  
End;  
  
procedure TQuadrado.SetX (Value : real);  
Begin  
  Self.FX := Value;  
End;  
  
{*****TRetangulo*****}  
procedure TRetangulo.Desenhe;  
Begin  
  showMessage('Retangulo' + FloatToStr(Self.X) + FloatToStr(Self.Y));  
End;  
  
function TRetangulo.GetY : real;  
Begin  
  Result := Self.FY;  
End;  
  
procedure TRetangulo.SetY (Value : real);  
Begin  
  Self.FY := Value;  
End;  
  
{*****TCirculo*****}  
  
procedure TCirculo.Desenhe;  
Begin  
  showMessage('Circulo' + FloatToStr(Self.Raio));  
End;  
  
function TCirculo.GetRaio : real;  
Begin  
  Result := Self.FRaio;  
End;  
  
procedure TCirculo.SetRaio (Value : Real);  
Begin  
  Self.FRaio := Value;  
End;  
  
end.
```

```
unit Vetor;

interface

const
  Tam = 100;
type
  Vetor_T = array[1..Tam] of integer;

  TVetorInteiro = class
  private
    FDados    : Vetor_T;
    FTamanho  : integer;
  protected
    function  GetDados (Posicao : integer) : integer;
    procedure SetDados (Posicao ,Value : integer);
    function  GetTamanho : integer;
    procedure SetTamanho (Value : integer);
  public
    property Tamanho : integer read GetTamanho write SetTamanho ;
    property Dados[Posicao : integer] : integer
      read GetDados write SetDados; default;
    Procedure Insere( Value : integer);
    procedure Inverte;

  End;

implementation

{ TVetorInteiro}
function TVetorInteiro.GetTamanho : integer;
Begin
  Result := Self.FTamanho;
End;

procedure TVetorInteiro.SetTamanho (Value : integer);
Begin
  Self.FTamanho := Value;
End;

function TVetorInteiro.GetDados ( Posicao : integer) : integer;
Begin
  Result := Self.FDados[Posicao];
End;

procedure TVetorInteiro.SetDados (Posicao , Value : integer);
Begin
  Self.FDados[Posicao] := Value;
End;
```

```
procedure TVetorInteiro.Insere( Value : integer);
Begin
  with Self do
    Tamanho := Tamanho + 1;
    try
      Dados[Tamanho] := Value
    except
      Tamanho := Tamanho - 1;
    end;
  end;
End;

procedure TVetorInteiro.Inverte;
Var
  i, f : byte;
  aux : integer;
Begin
  i := 1;
  f := Self.Tamanho;
  while i < f do
    Begin
      aux := Self[i];
      Self[i] := Self[f];
      Self[f] := aux;
      inc(i);
      inc(f,-1);
    End;
  end;
end.
```

Tratamento de Exceção

Aplicações Robustas

O tratamento de exceção é um mecanismo capaz de dar robustez a uma aplicação, permitindo que os erros sejam manipulados de uma maneira consistente e fazendo com que a aplicação possa se recuperar de erros se possível ou finalizar a execução quando necessário, sem perda de dados ou recursos. No object pascal os tratadores de exceções não tendem a obscurecer o algoritmo dos programas, assim como fazem a técnicas convencionais de tratamento de erros

Para que uma aplicação seja segura, seu código necessita reconhecer uma exceção quando esta ocorrer e respondê-la. Se não houver tratamento para uma exceção, será exibida uma mensagem descrevendo o erro. Uma exceção deve ser respondida sempre que houver perigo de perda de dados ou de recursos do sistema.

O Delphi possui uma classe especial para tratar exceções e quando o programador não define um código para tratar determinada exceção o Delphi irá utilizar um tratador default de exceções.

Os Recursos do Sistema que necessitam proteção e podem causar danos na execução caso sejam perdidos são: Arquivos, Memória, Recursos do Windows e Objetos.

Criação de um Bloco Protegido

```
try
  <instrução 1>;
  .
  .
  .
  <instrução n>
finally
  <instrução n + 1>;
  .
  .
  .
  <instrução m>
end;
```

- Tanto as instruções da cláusula *try* quanto as instruções da cláusula *finally* são executadas sequencialmente.
- Se houver uma exceção em qualquer das instruções da cláusula *try* o controle de execução será desviado para a primeira instrução da cláusula *finally*.
- As instruções da cláusula *finally* são sempre executadas, mesmo que não haja exceção.
- *try ... finally* não trata exceção em particular, apenas permite a garantia de que não haja perda de recursos.

Exemplos:

```
var
  F : File;
begin
  AssignFile(F, 'Arquivo.xxx');
  Reset(F);
  try
    { código que atualiza o arquivo F}
  finally
    CloseFile(F);
  end;
end;
```

Tratando Exceções

```
try
  <instrução 1>;
  .
  .
  .
  <instrução n>
except
  <lista de tratamentos>
[else
  <instrução 1>;
  .
  .
  .
  <instrução n>]
end;
```

Tratando Uma Exceção

```
on <tipo de exceção> do
  <instrução>;
```

- Quando uma exceção é tratada, o fluxo de execução continua na instrução seguinte ao tratamento.
- Para permitir que o tratamento da exceção prossiga no tratamento default, utiliza-se a palavra-chave *raise* no fim do bloco de instruções do tratamento para que a exceção seja lançada novamente.

Principais Classes de Exceção

Classe	Descrição
EAccessViolation	Acesso inválido a uma região de memória.
EConvertError	As funções StrToInt ou StrToFloat não consegue converte uma string num valor numérico válido
EDivByZero	Divisão de inteiro por zero
EGPFault	Acesso a memória não permitido
EInOutError	Erro de I/O de arquivo
EIntOverFlow	Operação inválida com número inteiro
EInvalidCast	Typecast inválido utilizando o operado As
EInvalidOp	Operação inválida com número real
EOutOfMemory	Memória Livre insuficiente para alocar objeto
EOverflow	Número real excedeu a faixa válida
ERangeError	Valor excede a faixa definida para o inteiro
EStackOverflow	Quando a pilha (stack) não pode crescer dinamicamente
EUnderflow	Número real menor que a faixa válida
EZeroDivide	Divisão de real por zero

Exemplos:

```
function Media(Soma, NumeroDeItens: Integer): real;
begin
  try
    Result := Soma div NumeroDeItens;
  except
    on EZeroDivide do
      Result := 0;
  end;
end;
```

Criação de Classes de Exceção

```
<identificador> = class(Exception);
```

- Qualquer anormalidade na execução de um programa pode ser transformada em uma exceção pela criação de uma classe para fazer esse tratamento.
- As classes de exceção devem ser subclasses da classe *Exception*.
- No momento em que houver uma anormalidade, um objeto do classe de exceção criada deve ser lançado, seguindo o modelo:

```
if <condição anormal> then  
  raise <classe de exceção >.Create( <descrição> );
```

Exemplos:

```
type  
  EFatorialNegativo = class(Exception)  
  public  
    constructor Create;  
  end;  
  .  
  .  
  .  
constructor EFatorialNegativo.Create;  
begin  
  inherited Create('Fatorial Negativo');  
end;  
  
function Fatorial(const n: Longint): Longint;  
begin  
  if n < 0 then  
    raise EFatorialNegativo.Create;  
  if n = 0 then  
    Result := 1  
  else  
    Result := n * Fatorial(n - 1);  
end;
```

```
var
  N: Longint;

begin
  Write('Digite um valor (-1 para sair): ');
  Read(N);
  while N <> -1 do
  begin
    try
      Writeln(n, '! = ', Fatorial(N));
    except
      on E:EFatorialNegativo do
        Writeln(E.Message);
      end;
    Write('Digite um valor (-1 para sair): ');
    Read(N);
  end;
end.
```

