Programação Orientada a Objetos com Object Pascal

João Morais

2003

SUMÁRIO

PARTE I
Conceitos da Programação Orientada a Objetos

1. Introdução

A OOP (Object Oriented Programming) revela uma forma distinta de estruturarmos nossos programas. Toda informação é colocada dentro de objetos, de forma organizada, e o uso ou a manutenção destas informações deve seguir as regras descritas pela classe do objeto em questão.

Vamos a alguns exemplos práticos: a tela de login, o botão Ok, a caixa de texto em que você digita a sua senha, são exemplos de objetos do seu programa. Por outro lado, o conceito de formulário (a tela de login), o conceito de botão (botão Ok) e o conceito de caixa de texto (caixa aonde se digita a senha) são exemplos de classes.

Há também classes de objetos não visuais, mas que precisam existir para que o programa possa funcionar, tal como a própria aplicação, uma tabela (arquivo de um banco de dados), entre outros.

Neste ponto você já consegue fazer uma distinção entre classes e objetos. Enquanto as classes definem a estrutura, as regras e os tipos de dados que serão armazenados, os objetos são os dados que o programa irá manipular. As classes são o conceito, os objetos são a informação.

A Programação Orientada a Objetos facilita a construção de grandes sistemas, pois o divide em partes bem definidas. Uma destas vantagens é a reutilização de código através de herança. Desta forma é possível que uma equipe com várias pessoas possa dividir um grande projeto, e cada parte deste projeto possa ser unida com as outras, sem que haja problemas com incompatibilidade.

2. Conceitos da programação orientada a objetos

Neste capítulo vou apresentar os conceitos da OOP. Use-o para referências futuras. Ao longo desta apostila, vários conceitos voltarão a ser abordados.

2.1. Conceitos básicos

2.1.1. Classe

É o conjunto de informações que modelam um objeto. A classe não ocupa espaço em memória e não tem dados específicos do programa. Ela possui apenas as regras que irão definir o comportamento de um objeto.

Exemplos de classes: o conceito de formulário, de botão, de caixa de texto e de tabela de banco de dados.

2.1.2. Objeto

É a informação em memória, e segue as regras definidas pela classe que o criou. Ao criar uma classe, você cria diversas regras que definem o comportamento de parte do seu programa, e a partir destas regras, você pode criar objetos, que irão armazenar estas informações.

Exemplos de objetos: ao criar regras que explicam como um botão ou uma tabela de banco de dados deve funcionar (as classes), pode-se criar objetos que irão armazenar as informações do programa, tal como o comportamento que determinado botão terá ao ser clicado, e a localização da tabela do banco de dados.

2.1.3. Instância

Instância é um sinônimo de objeto. Quando você cria um botão Ok para o seu formulário, costuma-se dizer que você cria uma instância da classe Botão. Todo objeto precisa ser criado, precisa ser “instanciado” para que possa ser utilizado.

2.1.4. Herança

É um conceito muito importante da programação orientada a objetos, e trata da reutilização do código. É bastante comum existirem classes de objetos com características em comum, e o conceito de herança, além de evitar a repetição do código, ajuda na depuração do programa e na criação de classes mais complexas e funcionais.

Um bom exemplo já abordado - as classes que criam objetos visuais (botões, formulários, etc.), e as classes que criam objetos não visuais (aplicação, tabela do banco de dados, etc.). Todos os objetos visuais possuem informações em comum, tal como as coordenadas de posicionamento na tela, tamanho, etc. Estas informações podem ser agrupadas em uma classe genérica e passadas às classes-filha através do conceito de herança.

2.1.5. Polimorfismo

Uma importante característica da programação orientada a objetos, e permite que determinada rotina, em uma classe genérica, possua implementações distintas em classes herdeiras.

Exemplo de polimorfismo: o botão e a caixa de texto precisam de uma rotina que faça o seu desenho na tela. Mas como utilizar uma rotina para criar desenhos distintos? O polimorfismo cuidará disto.

Neste ponto você me pergunta - se eu tenho desenhos distintos em classes distintas, eu posso simplesmente abandonar o conceito de herança e o conceito de polimorfismo, e criar rotinas separadas em cada classe. Mas aí eu te respondo - o polimorfismo e a herança são os grandes destaques da programação orientada a objetos, eles simplificam e muito a construção de sistemas complexos. A segunda parte desta apostila dará ênfase a estes conceitos.

2.2. Quanto ao conteúdo da classe

2.2.1. Atributos

São as informações que podem ser gravadas em um objeto, ou seja, são as variáveis declaradas na classe.

2.2.2. Propriedades

É possível criar uma estrutura que funcione como uma variável (recebendo e retornando dados), e controlada por um par de rotinas. Em uma propriedade, pode-se definir um procedimento que irá receber a informação, e/ou definir uma função que irá retornar a informação. Propriedades são especialmente úteis para proteger as informações dentro do objeto e/ou executar alguma instrução quando a informação é alterada ou recuperada.

Propriedades são definidas com a palavra reservada *property* à frente do nome da propriedade.

2.2.3. Métodos

São as rotinas da classe, e podem ser dos seguintes tipos:

2.2.3.1. Métodos Construtores (Constructor)

É o método executado no momento de criação da instância da classe. Dentro da sua declaração, pode-se colocar a inicialização de variáveis, entre outros processos que prepara o objeto para ser utilizado. Outra atribuição aos métodos construtores é ligar o objeto recém criado aos demais métodos deste objeto.

Toda classe deve ter ao menos um método construtor, e apenas um deve ser executado, uma única vez, antes de qualquer outro.

Métodos construtores são definidos com a palavra reservada *constructor* à frente do nome do método.

2.2.3.2. Métodos Destrutores (Destructor)

É o método executado para destruir o objeto e liberar a memória até então alocada.

Toda classe precisa de um e apenas um método destrutor. Este deve ser o último método a ser executado.

Métodos destrutores são definidos com a palavra reservada *destructor* à frente do nome do método, e devem ser declarados como sobrescritos.

2.2.3.3. Métodos Estáticos (Static)

Este é o tipo de método tradicional. Ele é declarado como procedimento ou função e não recebe qualquer sufixo. A localização deste método é definido em tempo de compilação (vide métodos Dinâmicos e métodos Virtuais).

2.2.3.4. Métodos Dinâmicos (Dynamic)

Métodos dinâmicos são aqueles ligados em tempo de execução, conforme a classe do objeto que o chamou.

Para que se faça valer o Polimorfismo, é necessário um tratamento diferenciado para métodos semelhantes - enquanto classes genéricas criam a chamada para determinada tarefa, as classes herdeiras implementam esta tarefa. Para que isto funcione corretamente, o método na classe genérica deverá ser dinâmico.

Métodos dinâmicos são definidos com a palavra reservada *dynamic*, logo após a declaração completa do método.

2.2.3.5. Métodos Virtuais (Virtual)

Tem funcionamento idêntico aos métodos dinâmicos, exceto que, métodos dinâmicos possuem uma implementação que favorece o tamanho do código, e métodos virtuais favorecem o tempo de execução. Por este motivo, procure sempre declarar seus métodos não-estáticos como Virtuais.

Métodos virtuais são definidos com a palavra reservada *virtual* logo após a declaração completa do método.

2.2.3.6. Métodos Sobrescritos (Override)

Os métodos sobrescritos são as implementações de métodos dinâmicos ou virtuais.

Criar um método dinâmico ou virtual em uma classe genérica é um pré-requisito para criar métodos sobrescritos nas classes herdeiras. Os métodos sobrescritos irão sobrepor as implementações herdadas, porém, de forma inteligente. Com esta técnica é possível utilizar uma classe genérica para chamar um método implementado em uma classe herdeira.

Este conjunto de métodos (dinâmico, virtual e sobrescrito) é o grande segredo para trabalhar com herança, e especialmente com polimorfismo. Este assunto será abordado com exemplos ao longo desta apostila.

Métodos sobrescritos são definidos com a palavra reservada *override*, logo após a declaração completa do método.

2.2.3.7. Métodos Abstratos (Abstract)

São métodos cuja implementação será feita apenas através dos herdeiros da classe. Utiliza-se em conjunto com métodos virtuais ou dinâmicos.

Pode ser necessário criar um método tão genérico em uma classe, que ele sequer possui implementação, sequer possui um código. Para não criar um método que simplesmente possui um begin seguido de um end, declara-se o método como abstrato e dispensa-se a necessidade de implementação.

Importante: não se deve criar instância de classes com algum método abstrato. A instância deverá ser criada através de uma classe herdeira que tenha todos estes métodos devidamente implementados.

Métodos abstratos são definidos com a palavra reservada *abstract*, logo após a declaração Dynamic ou Virtual.

2.3. Quanto à área da declaração

2.3.1. Privado (Private)

A área privada de uma classe contém declarações que não estão acessíveis, nem fora da classe, nem para seus herdeiros. Útil para guardar informações que precisam ter o conteúdo preservado ou protegido.

Uma área privada é iniciada com a palavra reservada *private*.

2.3.2. Protegido (Protected)

Área cuja declaração é visível apenas pela própria classe e seus herdeiros. Não é visível fora da classe.

Uma área protegida é iniciada com a palavra reservada *protected*.

2.3.3. Público (Public)

Indica declarações visíveis em qualquer local, seja dentro ou fora da classe.

Uma área pública é iniciada com a palavra reservada *public*.

2.3.4. Publicado (Published)

Declaração pública de propriedades e eventos. É utilizado em programação visual, e indica o que está visível para o programador em tempo de projeto. Útil para programação de componentes, portanto, está fora do escopo desta apostila.

Uma área publicada é iniciada com a palavra reservada *published*.

PARTE II
Implementação dos conceitos em Object Pascal

3. Introdução

A primeira implementação do Turbo Pascal orientado a objetos ocorreu em sua versão 5.5. Isto foi em meados de 1989. É lógico que nem todos estes conceitos existiam neste compilador, por exemplo, o conceito de propriedade e o conceito de classe, como existe nas versões mais modernas dos compiladores Pascal.

Esta apostila irá abordar a sintaxe do Free Pascal, do Delphi e demais compiladores compatíveis com estes.

4. Objetos e herança de objetos

Veja o exemplo abaixo:

```
type
  TForm_ok = Object
    Top, Left: Integer;
    Visible: boolean;
    constructor Create;
    destructor Destroy;
    procedure Show;
    procedure Hide;
    botao_ok: TButton;
  end;

var
  Form_ok: TForm_ok;
```

Neste exemplo está sendo declarado um formulário fictício. As declarações *constructor* e *destructor* indicam os procedimentos de abertura e encerramento do objeto.

Quero chamar a atenção para uma particularidade deste objeto - a redundância. Este é um exemplo bem simples, mas uma implementação completa de um formulário se estenderia a vários outros dados e métodos, seria totalmente inviável repetir tudo isto para cada janela que quiséssemos criar. Veja este segundo exemplo:

```
type
  TForm = Object
    Top, Left: integer;
    Visible: boolean;
    constructor Create;
    destructor Destroy;
    procedure Show;
    procedure Hide;
  end;

  TForm_ok = Object(TForm)
    botao_ok: TButton;
  end;

var
  Form_ok: TForm_ok;
```

Aqui foi utilizado um conceito da OOP - a herança. Temos, primeiramente, uma definição de tipo objeto - o TForm. O TForm é apenas uma estrutura que irá definir as características do objeto, e não o objeto em si. É como o Integer do Pascal, ele não guarda números, mas é utilizado na cláusula *var* para criar variáveis que guardem números inteiros.

Em seguida há uma nova definição de tipo objeto, que chamamos de TForm_ok. Mas afinal, qual a diferença entre eles? O primeiro, TForm, é um tipo genérico, contém as informações necessárias para criar um formulário vazio. O segundo, TForm_ok, herda todas as características do TForm e acrescenta um objeto botão.

Por último, cria-se uma variável, Form_ok. Este sim, é o nosso objeto. Dentro da estrutura do Form_ok há todas as informações de um formulário (posição, tamanho, etc.), o endereço em memória dos seus métodos (mostrar, esconder, destruir) que são ligados ao objeto pelo *construtor* e temos também as informações do botão, criado na implementação do tipo objeto TForm_ok.

5. Classes de Objeto e Instâncias

Vou colocar uma nova referência ao nosso exemplo:

```
type
  TForm = Object
    Top, Left: integer;
    Visible: boolean;
    constructor Create;
    destructor Destroy;
    procedure Show;
    procedure Hide;
  end;

  PForm_ok = ^TForm_ok;
  TForm_ok = Object(TForm)
    botao_ok: TButton;
  end;

var
  Form_ok: PForm_ok;
```

Ocorreram duas diferenças importantes neste exemplo. Note a declaração do tipo PForm_ok, e note também a nova declaração da variável objeto Form_ok. Vou falar da diferença entre objeto (citado no exemplo anterior) e classe (comentado no início da apostila). No primeiro exemplo, a variável Form_ok era o próprio objeto. Bastava, então, declarar esta variável (na área var) para que ela ocupasse o seu espaço na memória.

A classe é uma estrutura que aponta para o objeto alocado na memória. É um ponteiro para objeto.

Uma variável do tipo classe possui sempre o mesmo tamanho, independente do tamanho da sua estrutura. Isto é possível porque esta variável indica apenas a localização do objeto na memória. Mas então, para quê complicar tanto? Há alguns bons motivos para funcionar desta maneira, e o mais importante deles é que cria-se uma instância (leia-se - é roubada memória do computador) apenas quando for necessário, e depois a memória pode ser liberada ainda com o programa em funcionamento. Desta forma, vários programas podem compartilhar a memória do computador, harmônica e economicamente.

Tendo entendido este conceito de classe como um ponteiro para objeto (tenha certeza de tê-lo compreendido), vou repassar o que foi feito no último exemplo: PForm_ok = ^TForm_ok; - esta instrução cria um tipo "apontamento para o tipo objeto TForm_ok".

Vamos a outro exemplo:

```
var
  Form_ok: PForm_ok;

begin
  new(Form_ok, Create);
  Form_ok^.Show;
  dispose(Form_ok, Destroy);
end;
```

Aqui temos três clássicos exemplos de programação com alocação dinâmica: alocar (new), acessar (Form.Show) e liberar (dispose). O termo alocação dinâmica indica que foi reservado memória para os dados enquanto o programa estava em execução.

O procedimento new do Pascal aloca espaço na memória, suficiente para guardar a estrutura indicada. O primeiro parâmetro indica quem recebe o endereço da estrutura, e o segundo indica o método construtor.

Observação importante: nunca acesse um objeto que não foi instanciado. Na melhor das hipóteses, você será presenteado com uma mensagem de erro, nada amigável, informando uma violação de acesso à memória. No DOS, como não há proteção da memória, a sua sessão poderá ser encerrada sem maior cerimônia.

O conceito "ponteiro para objeto" continua valendo até hoje, mas é implementado de forma mais limpa e dedutível. Nosso último exemplo fica assim:

```
type
  TForm_ok = class(TForm)
    Botao_ok: TButton;
  end;

var
  Form_ok: TForm_ok;

begin
   Form_ok:=TForm_ok.Create(self);
  Form_ok.Show;
  Form_ok.Destroy;
end;
```

A palavra reservada *class* e o conceito de classe veio fazer companhia ao Object Pascal apenas no Free Pascal, no Delphi e em outros compiladores Pascal modernos. A única diferença estrutural da *class* para o ponteiro a objeto é ao referenciar métodos e atributos, em que é dispensado o uso do circunflexo - o compilador sempre saberá quando você está se referindo ao objeto ou ao seu endereço.

Não vá ficar com raiva de mim por ter feito você estudar ponteiros. Se você não entende como as coisas realmente funcionam, não poderá dizer que conhece o assunto ;-)

6. Herança, Polimorfismo e Métodos Não-Estáticos

Observe este outro exemplo:

```
type
  TControle = class(TObject)
    X, Y: integer;
    procedure Desenha; virtual; abstract;
  end;

  TBotao = class(TControle)
    procedure Desenha; override;
  end;

  TEdit = class(TControle)
    procedure Desenha; override;
  end;

procedure TBotao.Desenha;
begin
  // Métodos para desenho de um botão
end;

procedure TEdit.Desenha;
begin
  // Métodos para desenho de uma caixa de texto
end;
```

Agora, analise a seguinte situação: visualize uma tela qualquer do Windows ou qualquer outro ambiente, com diversos Rótulos (Labels), Caixas de Texto, Botões, etc. Você já sabe que cada um deles é um objeto, e que pertence a uma determinada classe. Também é fácil deduzir que estes desenhos não aparecem na tela por mágica. Cada objeto deve possuir um método responsável por seu desenho.

De forma simplificada, isto funciona assim: o sistema operacional ordena que determinada janela seja atualizada (redesenhada), então esta janela percorre a lista de controles que possui, e ordena que cada controle seja redesenhado. Esta lista de controles possui o endereço de cada objeto TControle.

Volte a observar o exemplo - há um método para cada um dos tipos de controle que foram implementados. Na tradicional programação procedural, a implementação fica assim:

```
if Controle is TEdit then
  DesenhaEdit
else if Controle is TBotao then
  DesenhaBotao;
```

Veja aqui a importância do polimorfismo na programação orientada a objetos: precisamos de um método virtual (abstrato, ou com alguma implementação padrão) que funcione como uma referência para o método correto. Veja agora, uma típica implementação orientada a objetos:

```
Controle.Desenha;
```

Internamente, ela funciona como o exemplo "procedural", colocado anteriormente. Perceba que Controle, por pertencer à classe TControle, pode referenciar tanto um TEdit como um TBotao. Como o método Desenha é virtual, o programa irá determinar, em tempo de execução, qual implementação será executada.

Para ajudar a ilustrar as explicações, verifique estes trechos de código:

```
var
  Controle: TControle;
```

```
begin
  Controle:=TBotao.Create;
  Controle.Desenha; // executa o método TBotao.Desenha
  Controle.Destroy;
  Controle:=TEdit.Create;
  Controle.Desenha; // executa o método TEdit.Desenha
  Controle.Destroy;
end;
```

7. Propriedades

Propriedades são muito úteis para manter a integridade das informações do objeto, de forma simples e transparente ao usuário da classe. Ao usuário, uma propriedade funciona como uma variável, por outro lado, permite ao programador da classe criar regras e ações que ocorrem no momento da leitura ou escrita dos dados.

Sintaxe da declaração de uma propriedade:

```
property <nome_propriedade>[ [<var_indice>] ]:<tipo>
 [read <função ou variável>]
 [write <função ou variável>]
 [default];
```

7.1. Uso dos parâmetros de uma propriedade

O *read* e o *write* de uma propriedade servem para modificar, respectivamente, a ação que deverá ser tomada no momento da leitura e da escrita do valor da propriedade.

Após a palavra reservada *read* ou *write*, pode-se utilizar um método ou uma variável. A variável poderá ser utilizada quando a leitura ou alteração do valor não requer qualquer tratamento ou ação, do contrário, se utiliza uma função (*read*) ou uma procedure (*write*)

Propriedades são tratadas como variáveis simples. Para leitura:

```
variavel:=obj.propriedade;
```

Para escrita:

```
obj.propriedade:=<valor ou variável>;
```

O uso de colchetes após a declaração do nome, trata a propriedade como uma matriz. O valor passado pelo usuário é enviado ao primeiro parâmetro da procedure ou função.

O *default* é utilizado em propriedades com índice (colchetes), e a coloca como a propriedade padrão da classe. Caso o usuário faça uma referência à classe, utilize um índice e não indique uma propriedade específica, a propriedade *default* será utilizada.

7.2. Exemplos de uso de propriedades

```
TParser = class(TObject)
private
  VExpression: string;
  function GetCallExpression: Extended;
public
  property Expression: string read VExpression;
  property CallExpression: Extended read GetCallExpression;
end;
```

A variável VExpression não é acessível fora da classe porque não pode ter seu conteúdo alterado indiscriminadamente, mas sua leitura é permitida. Expression é uma propriedade que permite apenas a leitura da variável VExpression, enquanto CallExpression é uma propriedade que executa uma ação para determinar o resultado da expressão.

```
TParser = class(TObject)
private
  VExpression: string;
  procedure SetExpression(AExpression: string);
public
  property Expression: string read VExpression write SetExpression;
end;
```

As propriedades são úteis para promover alterações no objeto. No exemplo acima, ao fazer a leitura da propriedade Expression, a informação será retirada diretamente da variável VExpression. Ao fazer a escrita de um valor a esta propriedade, será chamada a procedure SetExpression que receberá o novo valor através do parâmetro AExpression, verificará a integridade da expressão e providenciará as alterações necessárias ao objeto, para possibilitar o cálculo do resultado da expressão.

```
TLista = class(TObject)
private
  VItems: TList;
  function GetItem(AIndex: integer): TItemLista;
public
  property Items[AIndex: integer]: TItemLista read GetItem; default;
```

A propriedade Items é somente leitura e é utilizada para acessar os itens da variável privada VItems. Para acessar esta propriedade:

```
ItemLista:=Lista.Items[i];
```

O conteúdo da variável ‘i’ é passado ao parâmetro ‘AIndex’ da função GetItem. A propriedade Items é *default*, portanto, a linha acima poderá ser escrita da seguinte forma:

```
ItemLista:=Lista[i];
```

8. Área da declaração de atributos e métodos

Ao desenhar a estrutura de uma classe, é conveniente manter dados e métodos internos tão reservados e escondidos quanto possível. Uma classe bem desenhada mantém os dados vitais em áreas restritas, e o acesso a estes dados é feito através de propriedades e métodos.

```
TForm = class(TObject)
private
  VTop, VLeft: integer;
  procedure SetTop(ATop: integer);
  procedure SetLeft(ALeft: integer);
public
  property Top: integer read VTop write SetTop;
  property Left: integer read VLeft write SetLeft;
end;
```

O exemplo acima esconde as variáveis VTop e VLeft de qualquer implementação fora do contexto da classe TForm. Há um bom motivo para ser implementado desta forma - ao alterar a posição do formulário, é necessário redesenhá-lo em outra posição. Por isto as variáveis VTop e VLeft estão em uma área privada.

```
TFuncao = class(TObject)
protected
  procedure Recalc;
public
  function CalcFuncao: Extended; virtual; abstract;
end;

TFuncaoInterna = class(TFuncao)
public
  function CalcFuncao: Extended; override;
end;

TFuncaoExterna = class(TFuncao)
public
  function CalcFuncao: Extended; override;
end;
```

No exemplo acima há um método declarado na área *protected* da classe TFuncao, ou seja, é acessível apenas aos herdeiros desta classe. O segundo, declarado como público, é um método virtual e deverá ser aperfeiçoado por seus herdeiros.

Neste exemplo, o método CalcFuncao possui implementações distintas mas possui uma rotina em comum, chamada Recalc. O método Recalc foi declarado como protegido porque é parte de uma rotina, e não há qualquer interesse exceto ser chamado através dos herdeiros da classe.

Documento: Programação Orientada a Objetos com Object Pascal

Original: João Morais
Tradução: -
Revisão: João Morais
Publicação: 27/12/2003

Object Pascal
A · R · Q · U · I · V · O
joaomorais.com.br/pascal