PARTE I

Programação Orientada a Objetos (com exemplos em Delphi)

Por Cláudio Roberto Martins
claudio@cinbesa.com.br

SUMÁRIO

1. PROGRAMAÇÃO ORIENTADA A OBJETOS COM OBJECT PASCAL

O assunto apresentado neste tópico é discutir os conceitos teóricos da orientação a objetos (OO) em uma linguagem de programação, neste caso, com Object Pascal, que é a linguagem base do ambiente Delphi. O Object Pascal permite uma abordagem orientada a objetos, que se baseia em três conceitos fundamentais: classes, herança e polimorfismo. Também são vistas características específicas do Delphi quando se está trabalhando com OO.

1.1 CLASSES E OBJETOS

Classe – Tipo de dados definido pelo programador, que possui um estado, uma representação e algumas operações ou comportamentos. Uma classe tem alguns dados internos (atributos) e algumas operações (métodos), na forma de procedimentos ou funções. Uma classe descreve características genéricas e o comportamento de uma série de objetos semelhantes. Classes são usadas pelo programador para organizar o código-fonte e pelo compilador para gerar o aplicativo.

Objeto – É uma instância de uma classe, ou seja, é uma variável (dinâmica) do tipo definido pela classe. Objetos são entidades reais, que quando o programa está em execução, ocupam parte da memória para sua representação interna.

1.1.1 Classes

Uma classe é declarada através do uso da palavra reservada *class*, conforme o exemplo a seguir.

```
Type
  MinhaClasse = class

  End;  // fim da definição da classe
```

Essa classe não é útil, pois não tem definições de atributos nem métodos. Um exemplo mais interessante é mostrado a seguir.

```
Type
  Data = class
           Mes, Dia, Ano: Integer;
           Procedure AjustaValores(m,d,a: Integer);
           Function AnoBis: Boolean;
  End;  // fim da definição da classe
```

Os métodos da classe precisam ser implementados, indicando que fazem parte da classe. Para esta finalidade, o Object Pascal adota a notação de pontos, conforme exemplo a seguir.

```
Procedure Data.AjustaValores(m,d,a: Integer);
Begin
      Mes := m;
      Dia := d;
      Ano := a;
End;

// na implementation
```

(continua)

```
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;
```

1.1.2 Objetos

Agora que temos uma classe definida, para usá-la devemos criar uma instância de um objeto.

Assim como qualquer outro tipo de dados definido pelo programador, devemos criar um variável do tipo da classe definida. Porém, nos modelos de objetos do Object Pascal, uma variável do tipo de uma classe é apenas um referência, ou indicador, para a localização de memória em que o objeto será armazenado.

As instâncias de objetos definidos pelo programador devem ser criadas manualmente através da chamada do método *Create*, que é um **construtor** (procedimento especial usado para alocar memória para novos objetos e inicializá-los). Assim como se deve instanciar, também se deve dispor de um objeto, e isso é feito chamando-se o método *Free*.

Os métodos *Create* e *Free*, correspondem ao **construtor** e **destrutor** da classe TObject do Object Pascal. Todos os objetos criados no Delphi são herdados automaticamente desta classe, por isso usamos esses métodos sem declará-los. A determinação de sub-classes e heranças será discutida posteriormente. Um exemplo de instanciação e uso da classe definida é mostrado a seguir.

```
Type
   Hoje: Data;

Begin
   Hoje := Data.Create;
   Hoje.Dia := 7;
   Hoje.Mes := 3;
   Hoje.Ano := 2002;
   //ou
   Hoje.AjustaValores(7,3,2002);
   Hoje.Free;
End;
```

CUIDADO: Um objeto só pode ser utilizado depois da sua instanciação (***Create***), e antes da sua desalocação (***Free***), senão ocorrerá uma exceção no seu aplicativo.

1.2 DECLARANDO CONSTRUTORES

Para instanciar um objeto a partir de uma classe, chamamos o método *Create*, porém é muito comum ser necessário realizar alguma inicialização antes de usá-lo. Como esse objetivo, o Object Pascal permite a definição de construtores para as classes definidas pelo usuário. Da mesma forma, pode-se definir destrutores, caso seja necessário realizar alguma limpeza de recursos antes do objeto ser destruído. Construtores são procedimentos declarados com a palavra-chave *constructor*, e destrutores declarados com a palavra-chave *destructor*, conforme

o exemplo a seguir. Com o uso de construtores e destrutores, o Delphi já aloca e libera memória para o objeto desejado, ou seja, realiza a tarefa do *Create* e do *Destroy*.

```
Data = class
  Mes, Dia, Ano: Integer;
  constructor Init(d,m,a: Integer);
  Procedure AjustaValores(m,d,a: Integer);
  Function AnoBis: Boolean;
End;
```

Um exemplo de código para o construtor é mostrado a seguir.

```
constructor Data.Init (d,m,a: Integer);
Begin
  Dia := d; Mes := m; Ano := a;
End;
```

Assim, ao invés de instanciar o objeto como:

```
  Hoje := Data.Create;
  Hoje.AjustaValores(8,3,2002);
```

Podemos fazer:

```
  Hoje := Data.Init(8,3,2002);
```

1.3 OCULTANDO INFORMAÇÕES

A abordagem orientada a objetos introduz o conceito de **encapsulamento**, que é a capacidade de "esconder" detalhes de implementação (abstração), como se uma classe fosse uma caixa preta onde somente uma parte é visível: a interface da classe, representada pela assinatura de suas operações (métodos) públicas.

O principal objetivo do *encapsulamento* (o mesmo conceito de *hiding*, em inglês) 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 (programador) precise apenas conhecer a interface do objeto para poder utilizá-lo. Desta forma, o projetista poderá alterar tranqüilamente a implementação de um objeto (Classe) sem causar transtornos aos programas que as usam.

O ideal é fornecer acesso aos dados das classes sempre através de métodos, que saberão operar corretamente com eles. Imagine no exemplo da classe *Data*, se o programador atribuísse diretamente um valor ao atributo *Dia*, poderíamos chegar a um valor de data inexistente. Um método para ajustar os valores da data faria essa validação, e tiraria essa complexidade dos usuários da classe.

1.3.1 Operadores de visibilidade

O Object Pascal tem três especificadores de acesso básicos:

> **Private** (privado) – o membro (atributo ou método) é invisível fora da unit ou programa onde a classe é declarada. Em outras palavras, um método privado não pode ser chamado de outro módulo, e um atributo (propriedade) privado não pode ser acessado de outro módulo, também.

Public (público) – define que membros da classe (atributos e métodos) podem ser acessados livremente a partir de qualquer outra parte do programa.

Protected (protegido) – o membro é visível em qualquer parte do módulo (unit) onde a classe é declarada e de suas sub-classes (descendentes), independente da unit onde essas sub-classes forem declaradas.

O exemplo da classe Data pode ser modificado para:

```
Type
Data = class
  Private
      Mes, Dia, Ano: Integer;
  Public
      Procedure AjustaValores(d,m,a: Integer);
      Function AnoBis: Boolean;
  Protected
      Function DiasNoMes: Integer;
End;
```

Veja que agora foi adicionado o método **DiasNoMes** declarado como *protected*. Esse método retornaria o número de dias do mês atual. Esse método também poderia ser declarado como *private*, pois só faria sentido executá-lo no contexto da própria classe, mas como *protected* ela poderia ser usada a partir de uma classe herdada de Data.

No Delphi há um outro especificador de acesso, o *published*. Elementos desse tipo são acessados não só em tempo de execução, mas em tempo de projeto. Isso é usado para dar suporte à criação de componentes.

1.3.2 Property (propriedade)

Property (propriedades) representa um mecanismo, em Delphi, para encapsular os atributos (privates) de uma Classe, sobrecarregando as operações de leitura e escrita. São uma extensão natural às variáveis da instância (os membros atributos ou campos) de uma classe, pois permitem que o desenvolvedor pareça estar trabalhando diretamente com esses membros atributos, enquanto na realidade está executando chamadas a métodos.

Para utilizar **property,** os campos (atributos) devem ser declarados como *private*, os métodos como *protected*, e as propriedades (*property*) como *public*. A seguir, a declaração em Delphi.

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

onde,

Identificador representa a propriedade (property);

TIPO é o tipo ou classe da propriedade;

MétodoDeLeitura é o método (ou campo) associado à leitura da propriedade, correspondente a um método "get...."; e,

MédodoDeEscrita (ou campo) é o método associado à escrita da propriedade, correspondente a um método "set....".

No exemplo, a seguir, o atributo **Mes** é tratado pela *property* pMes, usando um método "setMes" para tratar a escrita do atributo.

```
Type
Data = class
  Private
     Mes, Dia, Ano: Integer;
  Public
     property pMes: integer read Mes write setMes;

  Protected
     procedure setMes(valor: integer);
End;

// na implementation:
procedure Data.setMes (valor: integer);
begin
   if (valor <= 0) or (valor >= 13) then
      // provoca um erro (exceção), alertando para o valor incorreto.
      raise Exception.Create ('Valor inválido para o Mes!')
   else
      Mes := valor;   // recebe o valor correto para mes.
end;
```

1.4 HERANÇA

1.4.1 Criando Subclasses

O Object Pascal permite que se defina um novo tipo de classe diretamente a partir de uma existente através de herança. Para herdar de um tipo existente, é preciso somente indicar o tipo da superclasse no início da declaração da subclasse, conforme o exemplo a seguir.

```
Type
  NovaData = class (Data)
     Public
           Function ObterTexto: String;
  End;
```

Neste exemplo, NovaData é derivada de Data. Podemos dizer que Data é ancestral de NovaData, ou classe-pai, e NovaData é descendente de Data, ou classe-filha.

A nova classe definida possuirá todos os atributos e métodos da classe ancestral e o novo método definido (ou redefinido, caso já exista) ObterTexto, que deve ser implementada, conforme a seguir.

```
Const
Meses: array[1..12] of String = ('Janeiro','Fevereiro','Março',...,'Dezembro');

Function NovaData.ObterTexto: String;
Begin
     ObterTexto := Format('%d de %s, %d, [Dia,Meses[Mes], Ano]);
End;
```

1.4.2 Herança e Compatibilidade de Tipo

Pascal é uma linguagem fortemente tipada, ou seja, não se pode atribuir a uma variável de um tipo um valor de outro tipo. No entanto, quando trabalhos com heranças em orientação a objetos, uma subclasse "é" do tipo da superclasse. Assim, pode-se utilizar um objeto de uma classe descendente quando um objeto da classe ancestral for esperado, mas o inverso não é verdadeiro.

A seguir é mostrado um exemplo com a classe Animal e uma classe Cao, herdada de animal.

```
Type
  Animal = class
      Public
            Constructor Create;
            Function ObterTipo: String;
      Private
            Tipo: String;
  End;

  Cao = class (Animal)
      Public
            Constructor Create;
  End;
```

Assim, a implementação dos métodos poderia ser:

```
Constructor Animal.Create;
Begin
  Tipo := 'Animal';
End;

Constructor Cao.Create;
Begin
  Tipo := 'Cao';
End;

Function Animal.ObterTipo: String;
Begin
  ObterTipo := Tipo;
End;
```

E poderíamos usá-los da seguinte forma:

```
Var
  UmAnimal: Animal;
  UmCao: Cao;
  MeuAnimal: Animal;

Begin
  UmAnimal := Animal.Create;
  UmCao := Cao.Create;
  MeuAnimal := UmAnimal;
  ShowMessage(MeuAnimal.ObterTipo);
  MeuAnimal := UmCao;
  ShowMessage(MeuAnimal.ObterTipo);
End;
```

Como dito anteriormente, a atribuição seguinte estaria errada:

```
      UmCao := UmAnimal;
```

1.5 POLIMORFISMO

É a capacidade de tratarmos objetos de diferentes tipos (classes) de uma mesma maneira desde que eles tenham um ancestral em comum.

Em outras palavras, 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.

1.5.1 Métodos Virtuais e Dinâmicos

Procedimentos e funções em Pascal baseiam-se em ligação estática, ou seja, o endereço da operação a ser chamada é resolvida em tempo de compilação e linkedição. Uma outra forma de ligação, usada em linguagens orientadas a objetos, é a ligação tardia ou dinâmica. Nessa forma, o endereço do método a ser executado é resolvido em tempo de execução.

Para esclarecer, suponha que uma classe (Animal) e sua subclasse (Cao) definam ambas um método, e que este tenha uma ligação dinâmica. Usando uma variável genérica (do tipo da superclasse) e aplicando o método definido, será chamado o método da superclasse ou da subclasse, dependendo do tipo do objeto atual atribuído à variável. Ou seja, o endereço do método será resolvido em tempo de execução.

A vantagem dessa abordagem se chama **polimorfismo**, e sua idéia básica é que se escreve a chamada de um método, mas o código chamado em tempo de execução corresponde ao tipo do objeto do método chamado.

No Object Pascal, métodos virtuais são definidos com a palavra reservada *virtual*, na superclasse, e a sua redefinição na subclasse, com a palavra reservada *override*. Um exemplo é dado a seguir usando o exemplo anterior e introduzindo um novo método **EmitirSom**.

```
Type
  Animal = class
    Public
        Constructor Create;
        Function ObterTipo: String;
        Function EmitirSom: String; virtual;
    Private
        Tipo: String;
  End;

  Cao = class(Animal)
    Public
        Constructor Create;
        Function EmitirSom: String; override;
  End;
```

Os métodos devem ser implementados na seção *implementation*, como a seguir.

```
Function Animal.EmitirSom: String;
Begin
  EmitirSom := 'Som do animal';
End;

Function Cao.EmitirSom: String;
Begin
  EmitirSom := 'Au Au Au';
End;
```

A criação de uma variável do tipo da superclasse (Animal) e chamada do seu método EmitirSom, deve funcionar se atribuímos qualquer objeto descendente da superclasse.

Veja que isso favorece muito a reutilização de código. Em vez de escrever uma chamada de método para cada tipo de objeto descendente, podemos atribuir o objeto a uma variável genérica e chamar seu método normalmente que a ligação será resolvida automaticamente.

O Object Pascal oferece uma variação desse mecanismo através do uso da palavra reservada *dynamic* ao invés de *virtual*. O efeito é o mesmo, a diferença é alguma variação em relação à como o compilador tratará a ligação dinâmica para fins de desempenho. Para maiores detalhes, consultar o *help* do Delphi. O Object Pascal também oferece um mecanismo que permite, ao invés de substituir totalmente o método da superclasse por uma nova versão, adicionar algum código a mais ao método existente. Isso é feito com a palavra reservada **inherited**, conforme exemplo a seguir.

```
Procedure Subclasse.Metodo;
Begin
  // Código adicional...
  // Chamada da versão do código da superclasse = Superclasse.Método.
  inherited;
End;
```

1.5.2 Classes e Métodos Abstratos

Uma classe abstrata é definida para ser base de uma hierarquia de classes. Possui métodos que não estão implementados e não pode ser instanciada.
O Object Pascal oferece um recurso que permite definir um método em uma superclasse e só implementá-lo em suas subclasses, são os **métodos abstratos**. A razão de métodos abstratos existirem é o **polimorfismo**: se uma superclasse possuir um método abstrato, toda subclasse poderá redefini-lo e continuamos a poder usufruir da ligação dinâmica, conforme discutido no item anterior.

No Object Pascal, é possível instanciar um objeto a partir de uma classe abstrata, porém, não é possível executar um método abstrato. Se isso ocorrer há uma exceção fatal e o aplicativo é finalizado. É difícil entender porque no Delphi é possível instanciar uma classe abstrata.

A definição de métodos abstratos se dá através do uso da palavra reservada *abstract*, conforme exemplo a seguir.

```
Type
  ClasseAbstrata = class
        Function Metodo: Integer; virtual; abstract;
  End;
```

1.5.3 Informação de Tipos em Tempo de Execução

Imagine que uma subclasse defina um método não presente na superclasse. Se tivermos uma variável do tipo da superclasse, e se esta referir-se à subclasse, será possível chamar o método definido anteriormente. Mas se a variável referir-se à superclasse, teremos um erro em tempo de execução.

Esse tipo de problema é comum, e o Object Pascal possui dois operadores (*is* e *as*) que nos permite lidar com tipos de classes em tempo de execução. O operador *is* retorna TRUE se um objeto é de um tipo de classe passado como parâmetro, conforme exemplo a seguir.

```
  If objeto is TipoClasse then ...
```

Assim, podemos saber com certeza o tipo de um objeto e podemos fazer um *typecasting* explícito sem causar erros de execução, como o exemplo a seguir.

```
  If Meu Animal is Cao then
        MeuCao := Cao(MeuAnimal);
```

Essa operação poderia ser realizada diretamente com o uso do operador *as*. Esse operador retorna um objeto do tipo da classe passada como parâmetro. A sua diferença para operador *is*, é que caso as classes não sejam compatíveis, é retornada uma exceção. Um exemplo do uso do operador *as* é dado a seguir.

```
MeuCao := MeuAnimal as Cao;
MeuCao.Comer;
```

Ou, simplesmente,

```
(MeuAnimal as Cao).Comer;
```

1.6 SOBRECARGA DE OPERADORES

O Object Pascal possui um recurso que permite a declaração de vários métodos com o mesmo nome, mas parâmetros diferentes, diferenciando a forma de uso, conforme as necessidades.

A sobrecarga de um método é feita com o uso da palavra reservada **overload**.

```
TMinhaClasse = class
  constructor Create; overload;
  constructor Create(msg: String); overload;
end;
```

Dependendo dos parâmetros usados na chamada do método, será realizado um comportamento diferente.

```
obj := TMinhaClasse.Create;
obj.Free;
obj := TMinhaClasse.Create('Ola');
obj.Free;
```

Métodos sobrecarregados são herdados pelas classes descendentes.

1.7 TÓPICOS AVANÇADOS

1.7.1 Self

A palavra reservada *self* refere-se a um parâmetro implícito chamado automaticamente a qualquer método quando ele é chamado. *Self* pode ser definido como um ponteiro para o objeto atual.

Como em um exemplo visto anteriormente, a implementação de um método deve operar somente sobre seus próprios atributos e não afetar os demais objetos. Procedure Data.AjutarValores (d, m, a: Integer);

```
Begin
  Dia := d;
  Mes := m;
  Ano := a;
End;
```

Nesse exemplo, **Mes** refere-se ao atributo do próprio objeto. Isso poderia ser expresso como:

```
Self.Mes := m;
```

1.7.2 Métodos de Classe

O Object Pascal permite a declaração de **métodos de classe**. Um método de classe é um método que não pode acessar os atributos de um objeto, mas pode ser chamado ao se referir a uma classe ao invés de uma instância. Tecnicamente, um método de classe é um método que não tem o parâmetro ***self***. Para declarar um método de classe, basta definir um método com o uso da palavra reservada **class**, conforme exemplo a seguir.

```
Type
  MinhaClasse = class
  …
  class function Metodo: Integer;
  …
end;
```

Esse tipo de construção pode ser usado para manter informações gerais relacionadas à classe, por exemplo, o número de objetos instanciados. Um exemplo de método de classe é o construtor **Create**, herdado de **TObject**. O método é usado sem que o objeto seja instanciado, afinal ele é responsável por realizar esta atividade.

```
Var
  Obj: MinhaClasse;
  I: Integer;
Begin
  I := MinhaClasse.Metodo;
  Obj := MinhaClasse.Create;
  Obj.Free;
End;
```

1.7.3 Ponteiros de Método e Tipos Procedurais

No Object Pascal existe o conceito de tipo procedural, que é parecido com o conceito de ponteiro para função em C. A sua declaração deve indicar a lista de parâmetros e retorno , se for uma função. Tipos procedurais são compatíveis com procedimentos que possuem exatamente os mesmos parâmetros. Dentro da abordagem orientada a objetos, o Object Pascal possui o mesmo recurso de tipos procedurais, só que se referindo a um método de um objeto, o que é chamado de ponteiro de método.

A declaração de um tipo ponteiro de método e tipo procedurais são semelhantes, a diferença é que para métodos são utilizadas as palavras reservadas *of object*, conforme o exemplo a seguir.

```
Type
  TipoProc = procedure(Par: Integer);
  TipoMetod = procedure(Par: Integer) of object;
```

Uma vez declarado o tipo, podemos criar um atributo deste tipo em um objeto:

```
Type
  MinhaClasse = class
    Atrib: Integer;
    Metodo: TipoMetod;
  End;

  OutraClasse = class
     Procedure Adiciona(N: Integer);
  End;
```

Após instanciar os objetos poderíamos fazer a atribuição (repare que os parâmetros das operação são exatamente iguais):

```
MeuObjeto.Metodo := OutroObjeto.Adiciona;
```

O conceito de ponteiros para métodos é a base para a construção de componentes com seus eventos, e corresponde ao conceito de **delegação**, na orientação a objetos. Lembre-se que no ambiente do Delphi estamos atribuindo nossos procedimentos aos eventos definidos pelo componente.

1.8 MANIPULANDO EXCEÇÕES

Exceção é um mecanismo que tem como objetivo tornar programas mais robustos ao adicionar a capacidade de manipular erros de software ou hardware. Permite o tratamento e reportagem de erros ou situações indesejáveis no programa com uma escrita de código mais compacto. Permite também, que se separe o código que descobre uma situação de erro do código que o trata e reporta.

No Object Pascal, toda exceção é um objeto do tipo de uma classe herdada de **Exception**. É possível criar novas exceções, simplesmente definindo uma subclasse de **Exception**.

O mecanismo de tratamento de exceções do Delphi baseia-se nas seguintes palavras-chave:

Try – Determina o início de um bloco de código protegido.

Except – Delimita o final de um bloco de código protegido e insere as instruções de manipulação:

```
On (tipo de exceção) do (instrução)
```

Finally – Indica um bloco de código opcional usado para desalocar recursos do bloco *try* que sempre será executado antes da manipulação da exceção.

Raise – Instrução usada para criar uma exceção definida pelo desenvolvedor, ou propagar qualquer exceção, inclusive emita pelo sistema, para o próximo manipulador.

Um exemplo de código para o tratamento de exceções é dado a seguir.

```
Function Divide(A, B: Integer): Integer ;
Begin
  Try
     Divide := A div B; //Pode gerar uma exceção de divisão por zero
  Except
     On EdivByZero do
     Begin
           Divide := 0;
           MessageDlg('Divisão por zero capturada', mtError, [mbOk], 0);
     End;
  End;
End;  // fim da function.
```

Outra forma de tratar uma exceção é obter o seu manipulador, ou objeto exceção passado pela instrução *raise*, quando esta for capturada. Com o manipulador, podemos acessar os seus atributos como um objeto qualquer. Um exemplo é dado a seguir.

```
On E: Exception do // Captura qualquer exceção
MessageDlg(E.Message, mtError, [mbOk], 0);
```

Um exemplo de criação de uma nova exceção é mostrado abaixo.

```
Type
      EminhaExcecao = class(Exception);
```

Para lançar essa exceção, no código da detecção do erro, basta usar raise com o método Create da classe de exceção desejada, conforme a seguir.

```
raise EminhaExcecao.Create('Mensagem de erro');
```

1.9 A BIBLIOTECA DE COMPONENTES VISUAIS

A biblioteca do sistema Delphi é chamada Biblioteca de Componentes Visuais (VCL), que inclui todos os componentes disponíveis para facilitar a criação de aplicações.

Todo o conjunto de classes da VCL segue uma hierarquia, onde TObject é o pai de todas as classes. Isto significa que todos os componentes e classes que criamos são herdados de TObject. Isso é feito implicitamente.

Se você notar o código-fonte de um formulário, você perceberá a estrutura de uma classe. O uso de componentes, como um formulário, é feito como o uso de qualquer outro objeto.

Os componentes possuem atributos, propriedades e métodos. Possuem também, ponteiros para métodos, que são atribuições que fazemos em tempo de projeto para definirmos eventos.

Componentes podem ser visuais, controles de interface com usuário, e não visuais. Muitos estão disponíveis na barra de componentes, mas outros, como o Tapplication e o TForm, não.

Além do componentes, a VCL também possui outros tipo de classes que não se enquadram nessa categoria. Basicamente, são classes para objetos gráficos (Ex. Tcanvas, Tbitmap, Tfont e outros), objetos de fluxo/arquivos (Ex. TfileStream, Treader, Twriter, etc) e Coleções (Ex. Tlist, Tstrings, Tcolletion e outros). Uma terceira parte da VCL é composta das classes de exceção. Classes de exceção são subclasses da Exception, descendentes diretas de TObject. O Help do Delphi explana detalhadamente toda a VCL, para maiores informações consultá-la.

Diante do exposto, a orientação a objetos é a base para a construção de seus próprios componentes, biblioteca de classes e exceções.

2. BIBLIOGRAFIA

1. Cantú, Marco – Dominando o Delphi 5, A bíblia – Makron Books.
2. Borland – Object Pascal Language Guide – Disponível on-line em: http://www.borland.com/.
3. Bobatti, Isaias – Programação Orientada a Objetos Usando Delphi – Visual Books.

3. EXERCÍCIOS

Exercício 1 – **Polimorfismo.**

Dê exemplos do uso do conceito de polimorfismo, em Delphi.

Exercício 2 **– Calculadora**

Modele uma classe para representar uma calculadora aritmética simples, que permita a realização das quatro operações básicas. Em seguida, elabore uma aplicação para manipular um *objeto* da classe calculadora, a partir de dois operandos, mostrando o resultado, seguindo a mesma metáfora do aplicativo Calculadora (padrão), do Windows.

Exercício 3 **– Classe Data**

Defina e implemente uma classe simples de data que verifique se o ano é bissexto, conforme exemplos da apostila, permita incrementar e decrementar o dia atual e tenha uma função que retorne a data.

Faça uma interface que instancie e permita o uso da classe, de forma a testá-la.

Exercício 4 **– Herança de Data**

Refaça o exemplo anterior com uma nova classe herdada de Data que redefina o método criado para retorno da data em formato textual com o nome dos meses.

Exercício 5 **– Herança e compatibilidade de tipo**

Implemente as classe Animal e Cao (ver apostila). Em um formulário, acrescente dois RadioButtons, um para cada tipo, e um botão, de forma que ao pressioná-lo, se execute o método ObterTipo do objeto correspondente ao radiobutton selecionado. Mostre o resultado no formulário.

Exercício 6 **– Métodos Virtuais**

Altere o exercício anterior acrescentando o método virtual EmitirSom (conforme apostila) e teste-o.

Exercício 7 **– Métodos Abstratos**

Altere o exercício anterior modificando o método EmitirSom para abstract na superclasse(conforme apostila) e teste-o.

Exercício 8 **– Informação de tipo**

Altere o exercício anterior de forma a testar o tipo de objeto antes de executar o método EmitirSom (conforme apostila) e teste-o.

Exercício 9 **– Figuras Geométricas**

1. Defina uma superclasse relativa a figuras geométricas e três subclasses relativas às especializações: círculo, quadrado e triângulo equilátero. Defina atributos e métodos de forma a permitir atribuição de valores, cálculo de perímetro, área e desenho das figuras na tela. Para isso, utilize o componente TpaintBox do Delphi (Ver help do Delphi).
2. Implemente um formulário no Delphi para a realização das operações definidas com a figura geométrica desejada.
3. Faz parte do exercício a correta definição de atributos e métodos às classes, e o melhor uso possível dessas dentro do paradigma OO.

Anexos

4. ANEXO 1 – RESUMO DA LINGUAGEM OBJECT PASCAL (DELPHI)

O item entre **[** e **]** é opcional.

Estrutura de uma Unit

```
Unit <Nome_Unit_identificador>;

Interface
      {Especifica o que será exportado pela UNIT a fim 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]        // Opcional
      {Código executado automaticamente quando um aplicativo que utiliza a
      UNIT é executado}
      <instruções>]

[Finalization]          // Opcional
      {Código executado automaticamente quando um aplicativo que utiliza a
      UNIT é finalizado}
      <instruções>]

end.
```

Classes

Definição de uma Classe

```
      <NomeClasse_identificador> = class[(Classe_Ascendente)]
            <atributos e métodos>
      end;  // fim da classe
```

- Deve ser feita na seção de declaração de tipos principal de um programa ou de uma unit, começando com a palavra reservada **Type**.
- 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 podem 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 da memória (ou "destruído").
- Se comporta como um ponteiro, mas é manipulado como uma variável normal.
- Pode ser atribuído o valor **nil (objeto NULO)**.

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

<Objeto> := Classe.Create;   {Instanciando}

<Objeto>.Free;               {Destruindo}

<Objeto>.Identificador       {Referenciando um membro ou método}
```

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 (Property)

- Representa um mecanismo para encapsular os campos de uma Classe sobrecarrgando as operações de leitura e escrita
- São uma extensão natural às variáveis de instância de uma classe, pois permitem que o desenvolvedor pareça esta 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, **MédodoDeEscrita** 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 Proprieda de>
```