

**Programação Orientada a Objetos
em Delphi**

Compilado com base em artigos de autoria de
Fabricio Mota (fmotaol@yahoo.com.br)

Sumário

Parte I – Programação Orientada a Objetos	3
Introdução	3
Mas <i>por que diabos</i> isso foi inventado?	3
Afinal de contas, o que é um Objeto?	4
Classes e Instâncias	4
Atributos	5
Construtores e Destruidores	6
Métodos	8
Parte II – Princípios: Herança	10
Generalização e Especialização	10
Herança	10
Declarando X, Instanciando Y	14
Herança Múltipla	15
Classes Concretas e Abstratas	15
Construtores e Destruidores específicos	16
Parte III – Princípios: Polimorfismo	18
Métodos Abstratos	23
Acessando os métodos originais	24
Métodos virtuais versus Métodos dinâmicos	25
Parte IV – Princípios: Encapsulamento	26
Mantendo uma vida particular	26
Classificando os membros por visibilidade	26
Propriedades: atributos a uma visão externa	29
Sobrecarga de Métodos	31
Visibilidade entre classes distintas	32
Público, privado ou protegido: quando devo declarar?	33

Parte I – Programação Orientada a Objetos

Introdução

Quantas vezes folheamos livros ou artigos de programação que falavam dessa tal “Programação Orientada a Objetos”, sem que entendêssemos sequer uma linha do que estava escrito? Muitas vezes. O que acontece é que esse par de letras “O” que tanto nos assusta, fica mais feio ainda depois de acompanhado de tantos palavrões, como “polimorfismo”, “herança múltipla”, “agregação” e outras *greguices* que insistem tanto em participar destes artigos.

Orientação a Objetos é o simpático nome que os pesquisadores da computação, na década de 70, usaram para batizar este novo “paradigma” que tinham acabado de inventar. E não é para menos, pois veremos que neste estilo de programação – como queriam os seus criadores – tudo é objeto. Ou ao mínimo, se pretende que seja.

Ainda como era nossa intenção falar, a Orientação a Objetos é um novo paradigma que procura re-arranjar a programação estruturada de uma maneira diferente, e talvez bem mais compreensiva ao raciocínio do homem. Ela pretende descrever a solução através de *objetos* – como no mundo real – ao contrário da programação estruturada, que o faz através de passos e comandos, semelhante a uma receita de bolo.

Um *paradigma* é uma maneira diferente de se enxergar uma mesma coisa, um mesmo mundo. O Paradigma OO busca bases filosóficas para tentar traduzir um mundo mais humano às instruções que o processador realmente entende. E é de *viagens* deste tipo, combinadas à estrutura de linguagens imperativas como o C ou Pascal, que surgem as linguagens OO. E estas últimas se responsabilizam pela existência da maioria dos grandes *softwares* produzidos nos dias atuais em todo o mundo.

Mas por que diabos isso foi inventado?

Desde que surgiu, a computação tem evoluído de uma maneira bastante rápida. Esta evolução acabou levando a comunidade deste meio a se deparar com a chamada “crise do *software*”. A orientação a objetos surgiu nos primórdios da década de 70, com as linguagens ADA, SIMULA e Small Talk. Ficou bastante difundida nos anos 80, com linguagens como o C++, e teve uma recente explosão em termos de divulgação nos anos 90, através da linguagem Java. Hoje em dia é muito comum encontrarmos compiladores e tradutores para linguagens OO, tais como Object Pascal, C++, Java e Visual Object.

A “crise do *software*” acabou levando os desenvolvedores da época a buscar uma solução alternativa, pois a complexidade dos sistemas já havia chegado a tal ponto que nem os seus programadores conseguiam dominá-la mais. Sabia-se que as linguagens estruturadas não davam mais suporte às necessidades, e esta possivelmente tenha sido a principal causa do surgimento das linguagens OO.

Também não é para menos. Se você tentar olhar um programa como o *AutoCAD* com os olhos de programador, irá entender o que queremos dizer com a palavra “complexidade”. Para um programador experiente em OO, fica difícil sequer imaginar uma solução destas sem que se modele um mundo de objetos. Daí a nossa intenção em dizer que a POO surgiu para suprir a crise do *software* – e conseguir dividir os sistemas

de maior complexidade em partes independentes que possam, assim, se manter sob o controle de seus criadores.

Afinal de contas, o que é um Objeto?

Os livros e artigos de OO costumam dizer que um objeto é *qualquer coisa*. Não que eles não tenham razão nesta frase, mas talvez não tenham muita clareza. Isto porque a resposta para essa pergunta deveria ser a mesma que você diria, se lhe perguntassem “O que é um objeto para você?”. Após responder a esta pergunta, basta tentar imaginar que você irá representar o mundo no seu programa através de objetos, e o significado começa a brotar na cabeça. Um objeto, para você, o que é? Um lápis? Um tijolo? Um copo? Exatamente.

Por definição, um objeto pode ser entendido como alguma coisa que *existe*, que possui *características*, e que *faz* algo. Este conceito é ideal para explicarmos o que é um objeto dentro do programa, em termos da linguagem. Um objeto, numa linguagem OO, é um *indivíduo* que possui atributos (estado) e métodos (comportamento), tal qual o mesmo objeto visto no mundo real. Os atributos, conforme veremos, representarão **dados** e os métodos, **trechos de código** executável.

Classes e Instâncias

Para entender melhor os princípios da OO é necessário abordar dois elementos essenciais: *classe* e *instância*. Estes elementos, que parecem estar mais ligados à filosofia do que à própria programação, são fundamentais aqui, mesmo com a mais simples das explicações.

Podemos dizer que classes são o conjunto de todos os elementos semelhantes, no nosso caso, objetos. Todo objeto pertence a uma determinada classe, assim na terra como no céu, isto é, assim no mundo real quando no “mundo” descrito no programa. Classes são *tipos* de coisas, e descrevem as coisas de uma maneira generalista, através de regras a que todos os elementos que a ela pertencem devem seguir, sem que necessariamente se conheça ‘uma ou outra’ dessas coisas.

Assim existe, por exemplo, a classe dos livros, a que pertencem todos os livros publicados. À classe dos equipamentos eletrônicos pertencem todos os computadores, televisores e outros dispositivos, bem como à classe dos humanos se espera que pertençam todas as pessoas.

Em contrapartida, citamos a instância, que significa um elemento específico daquela classe. Uma instância é um constituinte singular e conhecido da população daquela classe. A instância, ao contrário da classe, procura referir um elemento conhecido da sua classe.

Assim como você é uma instância da classe dos humanos, tal qual este monitor à sua frente é uma instância da classe de equipamentos eletrônicos, ou ainda aquele livro de Biologia de Soares disposto ao lado de sua cabeceira é uma instância da classe dos livros. Instâncias são, portanto, *casos* de ocorrência das classes.

Agora tratando novamente de classes, porém em termos de programação, diremos que elas são tipos declaráveis. No Delphi, a declaração de uma classe é feita através da palavra reservada *class*, como segue abaixo:

```
unit Unit1;
```

```
type
```

```
  TAnimal = class
```

```
  end;
```

Logo, TAnimal representará a classe dos animais, e nela serão definidas as características (regras) a que todo e qualquer animal deverá seguir. Vale lembrar que, por padrão no Delphi, convencionou-se os tipos como identificadores começados com a letra T.

Entretanto, embora tenhamos acabado de criar a classe dos animais, ainda não existe nenhum animal no nosso sistema. Isto porque não existe nenhuma *referência* a um animal, muito embora o compilador já “saiba” da possível existência dos animais. Para que possamos referenciar uma **instância** de animal, declararemos uma variável do tipo TAnimal:

```
var
```

```
  MeuAnimal: TAnimal;
```

Isto, teoricamente, resolve o nosso problema. Até porque agora já temos uma *referência* a um animal, apesar de não o conhecermos. Mais adiante veremos que isso não é exatamente verdade, mas vamos considerar que ele existe, por enquanto. Agora você sabe que possui um animal, e que se referiu a ele como **MeuAnimal**.

Atributos

Bom, você já declarou sua primeira classe. Mas o que ela faz? Para que ela serve? Absolutamente nada, por enquanto. Como sugerimos antes, um objeto é uma entidade que deve possuir estado e comportamento. E é para dar esta *vida* aos objetos que lhes caracterizamos com atributos e métodos. Nesta seção vamos explicar como dar *estado* ao objeto, ou seja, dar-lhe a capacidade de guardar informações.

Voltando ao nosso primeiro exemplo, iremos aperfeiçoar melhor a descrição do nosso animal, dando-lhe duas características, *altura* e *peso*. Também vamos personalizá-lo, dando-lhe a possibilidade de ter um *nome*. O nosso exemplo ficará assim:

```
unit Unit1;
```

```
type
```

```
  TAnimal = class
```

```
    Nome: string;
```

```
    Altura, Peso: real;
```

```
  end;
```

Bom, a partir de agora o seu objeto já possui a capacidade de guardar informações dentro dele. Poderíamos dar um nome, uma altura e um peso a `MeuAnimal`, acessando os seus dados internos (atributos) como no exemplo abaixo:

implementation

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MeuAnimal.Nome := 'Cachorro Plutão';
  MeuAnimal.Altura := 0.72;
  MeuAnimal.Peso := 9.3;
end;
```

Entretanto, se você compilar o seu projeto, e mandar que ele execute este código, ele não deverá funcionar, levantando um erro de violação de acesso. Mas por que? Simples e óbvio. Você realmente ordenou ao sistema que ele atribuísse o nome, peso e a altura ao `MeuAnimal`, mas você esqueceu de dizer *quem* era ele. Em outras palavras, a sua referência `MeuAnimal` está apontando para ninguém. Em termos do Delphi, diz-se que ela é *nula*, possui valor **nil**.

Mas então para quem eu devo fazê-la apontar? Eis a questão. Acontece que não existe nenhum animal no seu sistema. Se você voltar alguns parágrafos deste texto, onde ainda falávamos do que vem a ser um objeto, se lembrará que um fundamento básico e filosófico do objeto é existir. E para que alguma coisa passe a existir, ela primeiro deve ser criada. E assim como no mundo real, através de chamadas específicas, faremos objetos serem construídos e destruídos no nosso sistema.

Construtores e Destruidores

Construtores e destruidores são *métodos* especiais – mais abaixo explicaremos melhor o que são métodos – responsáveis pela criação e destruição de objetos. Os especialistas em linguagens orientadas a objeto costumam dizer que eles são métodos *de classe*. Mas isso não vem muito ao caso aqui.

O que importa é que em algum momento do nosso exemplo, um animal precisa ser criado, até para que possa receber aqueles valores que tentamos lhe atribuir mais acima. E é justamente esse o papel que tem um construtor. O construtor de uma determinada classe seria um método ou procedimento capaz de criar, ou *instanciar*, um objeto pertencente àquela classe. Sendo assim, para que construamos o nosso animal, usaremos a seguinte linha de código:

```
MeuAnimal := TAnimal.Create;
```

Esta chamada irá invocar o construtor padrão da classe, derivado da classe ancestral de todas, a classe `TObject`. Mas não se preocupe com isso agora. O importante é que, a partir da execução desta linha de código, passa a existir uma instância de `TAnimal` no seu programa, com memória alocada para ela, e `MeuAnimal` agora aponta para esta instância. Tão logo, para que aquele nosso código que resultou em erro funcione, vamos refazê-lo da seguinte maneira:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    MeuAnimal := TAnimal.Create;  
    MeuAnimal.Nome := 'Cachorro Plutão';  
    MeuAnimal.Altura := 0.72;  
    MeuAnimal.Peso := 9.3;  
end;
```

Uma vez feito isto, e depois de executado este bloco de código, MeuAnimal passa a existir na memória, e a armazenará os dados que ora inserimos nele. Se não acreditar, ou se quiser ter certeza disto, execute o bloco de código seguinte:

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    ShowMessage('Meu nome é ' + MeuAnimal.Nome);  
    ShowMessage('Minha altura é ' + FloatToStr(MeuAnimal.Altura) + ' cm');  
    ShowMessage('Meu peso é ' + FloatToStr(MeuAnimal.Peso) + ' kg');  
end;
```

Agora que já sabemos o que é um construtor, passemos para o destruidor (ou *destrutor*, no vício da tradução). Um destruidor, como sugerido antes, é capaz de destruir o objeto em questão, liberando a memória que foi alocada para ele. MeuAnimal, portanto, deixa de existir a partir do momento que executamos este código:

```
procedure TForm1.Button3Click(Sender: TObject);  
begin  
    MeuAnimal.Destroy;  
end;
```

A área de memória reservada para aquele TAnimal foi liberada. Entretanto, é importante ressaltar que isto não limpa a referência MeuAnimal, que continua apontando para a região de memória onde *estava* o TAnimal. Esta inconsistência pode trazer conseqüências desagradáveis para o sistema. Portanto, para um código seguro, sempre *informe* às referências que elas não mais apontam para ninguém, imediatamente após destruir os objetos:

```
procedure TForm1.Button3Click(Sender: TObject);  
begin  
    MeuAnimal.Destroy;  
    MeuAnimal := nil;  
end;
```

Métodos

Conforme dissemos já algumas vezes, um método é um bloco de código executável, capaz de definir *comportamento* ao objeto. Ao grosso modo, um método pode ser entendido como um procedimento (*procedure*) ou função (*function*), porém com a diferença de que ele se limita – ou ao menos deve se limitar – ao escopo **do objeto** a qual ele pertence. Métodos, de uma maneira geral, estão vinculados às instâncias.

Não fugindo muito da prática, e como se era de esperar, voltemos ao nosso velho exemplo do animal. Já lhe demos a capacidade de possuir estado, guardando algumas informações. Agora lhe daremos a possibilidade de se comportar, ou seja, literalmente o animaremos. Acrescentaremos em TAnimal três métodos: Correr, Crescer e Encolher. Estes métodos lhe darão a possibilidade de mudar seu estado. Vejamos como fica a nossa declaração:

```
unit Unit1;
```

```
type
```

```
  TAnimal = class
```

```
    Nome: string;
```

```
    Altura, Peso: real;
```

```
    procedure Correr;
```

```
    procedure Crescer(const incremento: real);
```

```
    procedure Encolher(const decremento: real);
```

```
  end;
```

```
var
```

```
  MeuAnimal: TAnimal;
```

```
implementation
```

```
procedure TAnimal.Crescer(const incremento: real);
```

```
begin
```

```
  Altura := Altura + incremento;
```

```
  ShowMessage('Minha nova altura é ' + Altura);
```

```
end;
```

```
procedure TAnimal.Encolher(const decremento: real);
```

```
begin
```

```
  Altura := Altura - incremento;
```

```
  ShowMessage('Minha nova altura é ' + Altura);
```

```
end;
```

```
procedure TAnimal.Correr;  
var i: Integer;  
begin  
  ShowMessage('Eu, ' + Nome + ', estou correndo!');  
  for i := 1 to 5 do  
    begin  
      ShowMessage('Estou no meu passo de número ' + IntToStr(i));  
    end;  
  ShowMessage('Eu, ' + Nome + ', cansei de correr.');
```

E invocaremos os métodos conforme abaixo:

```
procedure TForm1.Button4Click(Sender: TObject);  
begin  
  MeuAnimal.Crescer(0.1);  
  MeuAnimal.Encolher(0.5);  
  MeuAnimal.Crescer(0.3);  
  MeuAnimal.Encolher(0.2);  
end;
```

```
procedure TForm1.Button5Click(Sender: TObject);  
begin  
  MeuAnimal.Correr;  
end;
```

Os métodos da classe TAnimal manipulam diretamente os atributos, fazendo com que o objeto dono do método invocado – ele e somente ele – seja manipulado. E é exatamente essa a proposta de definição de comportamento dos objetos, manipular dados dentro de seu escopo.

Muitos outros métodos são e devem ser desenvolvidos, de maneira a descrever, da forma mais adequada possível, o comportamento desejado do objeto diante do sistema. Entretanto, veremos mais adiante, numa próxima publicação, onde abordaremos outros conceitos da OO.

Parte II – Princípios: Herança

Generalização e Especialização

Voltemos a tratar de um objeto novamente como um elemento do mundo. Ele é definido por uma determinada classe e, uma vez construído, toma a sua forma concreta – isto é, passa a existir – como instância desta classe. Logo, este objeto é um elemento pertencente a esta classe. No exemplo anterior, definimos uma classe de animais chamada TAnimal, contendo alguns atributos e métodos. Ainda estabelecemos uma instância desta classe, a qual nos referimos por MeuAnimal.

Entretanto, surge que classes únicas e separadas entre si, como exemplificamos antes, não são suficientes para definir todo um universo e as coisas que nele existem. Isto acontece principalmente porque alguns objetos podem pertencer a mais de uma classe ao mesmo tempo. Na verdade, quase todos eles provavelmente pertencem. Podemos dizer que um determinado livro é uma instância da classe dos livros, enquanto o mesmo livro também pertence à classe dos materiais escolares. E por que não?

Neste caso, surge uma questão bastante comum no nosso cotidiano, no que diz respeito à classificação das coisas. Eis que a questão é: algumas classes são subconjuntos de outras. A classe dos livros pode ser vista como um subconjunto *especial* da classe dos materiais escolares. Em palavras mais sérias, diríamos que os materiais escolares **generalizam** os livros. Os livros, por sua vez, seriam uma **especialização** dos materiais escolares, se olharmos pelo sentido contrário.

Se pararmos para observar um pouco, existem muitas outras situações envolvendo este fato. As pessoas, por exemplo, são uma classe especial de mamíferos, estes que por sua vez são uma subclasse de animais, enquanto estes últimos são generalizados por seres vivos. Olhando ao próprio redor, podemos encontrar muitos outros casos bem parecidos.

E como resolver a esta questão? Eis que surge a *herança*, que é um recurso voltado a resolver este tipo de problema, trazendo ainda consigo uma série de outras vantagens que vêm a otimizar o desenvolvimento de soluções em código executável. Falaremos sobre ela a seguir.

Herança

Nada de muito extraordinário ao nosso entendimento será tratado aqui, uma vez que já temos alguma noção de generalização e especialização. Extraordinário com certeza é o poder que este recurso tem. A herança vem a ser a forma usada pelas linguagens OO para implementar a especialização. Uma determinada classe que especialize outra, dizemos que ela *herda* desta última. Assim dizemos que a classe herdada é a *ancestral*, e a herdeira é a *descendente*. Vejamos o exemplo abaixo, adaptado do que mostramos no anteriormente:

```
unit Unit1;
```

```
type
```

```
TSerVivo = class  
end;
```

```
TAnimal = class(TSerVivo)  
end;
```

```
TVegetal = class(TSerVivo)  
end;
```

A partir deste par de parênteses depois da declaração da classe TAnimal, fizemos esta classe *herdar* de TSerVivo. Em outras palavras, nosso sistema agora entende que todo animal também é um ser vivo, e o especializa. O mesmo se aplica a todo vegetal. E é assim que o nosso programa enxergará estes objetos: as instâncias de TAnimal e de TVegetal agora serão, todas ao mesmo tempo, pertencentes à classe TSerVivo, devendo todas serem tratadas como tal. Uma classe pode, por sua vez, ter inúmeras outras classes diferentes como descendentes.

E a classe TSerVivo, herda de quem? De ninguém? Excelente pergunta. E bem oportuna. Quando você não declara uma ancestral à sua classe, o Delphi automaticamente o faz descendente da classe mais ancestral de todas, a classe TObject. Agora vamos complicar um pouquinho mais o nosso modelo:

```
unit Unit1;
```

```
type
```

```
TSerVivo = class  
  Alimento: string;  
  Peso: Real;  
end;
```

```
TAnimal = class(TSerVivo)  
  Membros: Integer;  
  Velocidade: Real;  
end;
```

```
TVegetal = class(TSerVivo)  
  Fruto: string;  
end;
```

Agora, além da relação de herança entre as classes TSerVivo e as descendentes TAnimal e TVegetal, demos atributos a cada uma delas. Todo TAnimal agora possui uma velocidade de locomoção, representado pelo atributo Velocidade, bem como um número de membros. Os vegetais por sua vez terão o nome do fruto que produzem. De igual forma, demos a todo TSerVivo o direito de escolha por um alimento preferido, representado pelo campo Alimento, bom como uma informação de peso.

Agora chegamos num ponto interessante. Dada as seguintes referências:

```
var  
MeuSerVivo: TSerVivo;  
MeuAnimal: TAnimal;
```

Poderemos facilmente atribuir um número qualquer de membros a MeuAnimal, desde que ele esteja instanciado. De igual maneira, podemos informar a MeuSerVivo que ele gosta de comer 'frutas tropicais', simplesmente atribuindo-lhe o valor ao respectivo atributo. Mas podemos atribuir um peso a MeuAnimal? A resposta é uma outra pergunta: *Por que não?*

Como se pode imaginar, todo animal também é um ser vivo e, como ser vivo, também respeita às características e regras estabelecidas para todos os seres vivos. Em outras palavras, TAnimal literalmente *herda* os atributos de TSerVivo, podendo ser tão manipulável nisto enquanto TAnimal, como qualquer instancia de TSerVivo. E isto se aplica tanto aos atributos quanto aos métodos.

Não existe um limite máximo para quantos níveis de especialização se pode haver entre classes. Uma classe ancestral, tal como é TObject, pode ter milhares de descendentes em cascata, e isto é o que torna os sistemas OO dotados de um amplo poder de classificação.

Agora vamos incrementar nosso modelo um pouco mais:

```
unit Unit1;  
  
-  
type  
TSerVivo = class  
  Alimento: string;  
  Peso: Real;  
  procedure AlimentarSe(const Alimento: string);  
end;  
  
TAnimal = class(TSerVivo)  
  Membros: Integer;  
  Velocidade: Real;  
  procedure LocomoverSe(const Direcao: string);  
end;  
  
TVegetal = class(TSerVivo)  
  Fruto: string;  
  procedure ProduzirOxigenio;  
end;  
  
var  
MeuSerVivo: TSerVivo;  
MeuAnimal: TAnimal;  
MeuVegetal: TVegetal;
```

implementation

```
procedure TSerVivo.AlimentarSe(const Alimento: string);  
begin  
  if Self.Alimento <> Alimento then  
    ShowMessage('Estou comendo ' + Alimento + ', mas eu gostaria de estar' +  
      'comendo ' + Self.Alimento)  
  else  
    ShowMessage('Estou comendo ' + Alimento + ', que é o que eu gosto');  
    Peso := Peso + 1;  
end;
```

```
procedure TAnimal.LocomoverSe(const Direcao: string);  
var i: Integer;  
begin  
  for i := 1 to Membros do  
    begin  
      ShowMessage('Estou movendo meu membro No.' + i + ' para ir ao ' + Direcao);  
    end;  
  ShowMessage('Estou a ' + FloatToStr(Velocidade) + ' Km/h');  
end;
```

```
procedure TVegetal.ProduzirOxigenio;  
begin  
  ShowMessage('Estou contribuindo para a atmosfera do nosso planeta.');
```

end.

Ora, sabe-se que apenas os vegetais produzem oxigênio, e por isso declaramos o método ProduzirOxigenio apenas em TVegetal. Mas sabe-se que tanto os animais quanto os vegetais se alimentam, e por isso tivemos o cuidado de declarar o método AlimentarSe na classe imediatamente superior a elas.

Você perceberá que poderá invocar o método AlimentarSe tanto de MeuAnimal quanto de MeuVegetal, uma vez que ambos são instâncias de TSerVivo. Entretanto, vale notar que os elementos membros de TAnimal não são acessíveis por instancias de TSerVivo, já que seres vivos não são necessariamente animais.

Observa-se que a herança possui um grande papel, tanto na questão de modelar classes relacionadas, quanto na economia de trabalho dos programadores. Se considerarmos uma classe *B* qualquer herdada de uma classe *A*, teremos que *B* trará automaticamente para si *todas* as características (atributos e métodos) de *A*, não sendo necessário reescrevê-las.

Declarando X, Instanciando Y

Como vimos logo acima, um objeto pode ser instância de uma classe que é descendente de uma outra mais ancestral. Este objeto ainda pode ser tratado como uma instância desta última classe mais genérica. Tendo isto, o critério de generalização disposto pela herança nos permitirá uma boa flexibilidade quanto às referências. Vejamos:

var

```
MeuSerVivo: TSerVivo;
```

implementation

```
procedure TForm1.Button6Click(Sender: TObject);
```

```
begin
```

```
  MeuSerVivo := TAnimal.Create;
```

```
end;
```

Pode-se ver, o exemplo acima declara uma referência a um TSerVivo, e parece a contrariar, instanciando para ela um objeto do tipo TAnimal. Isto é permitido? Sim, é claro. Como já demonstramos, todo TAnimal também é um TSerVivo, herda todas as suas características, e não existe nenhuma razão lógica que proíba isso. Apenas a situação contrária, como já vimos, não seria permitida, já que nem todo ser vivo se trata de um animal.

O único inconveniente que isso provoca é o fato de que, uma vez a referência sendo do tipo da classe superior, não há como saber de imediato que a instância apontada é da classe inferior. Logo, os atributos e métodos exclusivos de TAnimal não estarão disponíveis em MeuSerVivo em tempo de compilação. Verifique isso digitando MeuSerVivo. [control + espaço], e você perceberá que os únicos atributos e métodos disponíveis são os declarados até a classe TSerVivo.

Mas isso não quer dizer que você não tenha como acessá-los definitivamente. Uma vez que você tem certeza que a instância apontada pela referência MeuSerVivo é do tipo TAnimal, você pode *informar* ao compilador que se trata de uma instância de TAnimal, fazendo um *typecasting* para esta classe. Desta forma, pode-se resgatar novamente os atributos e métodos de TAnimal. Para *typecasting* de classes usa-se o operador **as**:

```
procedure TForm1.Button6Click(Sender: TObject);
```

```
begin
```

```
  MeuSerVivo := TAnimal.Create;
```

```
  (MeuSerVivo as TAnimal).LocomoverSe('Leste');
```

```
end;
```

Feito isso, o compilador passa a “entender” que o objeto apontado por MeuSerVivo é do tipo TAnimal, confiando na palavra de honra do programador, e liberando assim os seus atributos e métodos para uso. **Nota:** ao fazer *typecasting*, certifique-se de que **realmente** o objeto apontado seja uma instância da classe declarada após o **as**, ou isto poderá resultar numa exceção em tempo de execução.

Herança Múltipla

Bom, agora vamos fugir um pouco ao assunto para tratar deste outro mais controverso. Herança Múltipla é um conceito um pouco mais complicado de herança nas linguagens OO, e bastante questionado entre os estudiosos da área. Sabe-se que a herança múltipla é implementada em certos pontos no Ansi C++.

A herança múltipla visa permitir que uma classe possa herdar características de **mais de uma classe ancestral ao mesmo tempo**, fazendo com que, por exemplo, um carro-anfíbio possa herdar todas as características das classes Carro e Barco ao mesmo tempo. Trata-se de um recurso bastante poderoso em termos de linguagem. Mas é igualmente perigoso, e possivelmente complicado no desenvolvimento do seu compilador.

O Delphi, como muitas das linguagens OO comerciais e amadurecidas, não implementa a herança múltipla. Por isso não entraremos em mais detalhes sobre esse assunto aqui. Mas tanto o Delphi quanto o Java usam de recursos que podem *simular* a herança múltipla em determinados aspectos que são benéficos – especialmente os relacionados à ‘tipagem’ e à abstração – e ao mesmo tempo isolar outros que podem ser perigosos e complicados – como a herança de atributos. A este “recurso” usado pelo Delphi e pelo Java, damos o nome de **interface**. Ainda não veremos isto nesta publicação, ficando para as posteriores.

Classes Concretas e Abstratas

Bom, em meio às dezenas – às vezes centenas – de classes existentes num modelo, notaremos que algumas dessas classes poderão ser instanciadas, outras não. Isto se deve ao fato de que, muito alguns objetos possam pertencer a uma classe ancestral, acontece de não haver como existir um elemento que represente, de forma física e real, a classe em questão.

Voltamos ao nosso exemplo de TAnimal e TSevivo. Muito embora nada no Delphi nos impeça de instanciar um TSevivo na sua pura essência, isso não faz o menor sentido. Você já viu algum ser vivo que não pertença a uma classe mais inferior?

A casos como este, em que não existe nenhuma razão para a construção de objetos especificamente de uma classe, chamamos esta classe de **classe abstrata**. Ao caso contrário, isto é, de classes instanciáveis, chamamos de classes concretas. Classes abstratas geralmente são ancestrais de outras classes concretas.

Infelizmente no caso do nosso exemplo só tratamos de classes abstratas – é claro que vamos desconsiderar o construtor didático invocado no exemplo anterior. Mas em termos conceituais, não faz sentido se instanciar um TAnimal. Se alguém lhe perguntasse “qual o seu bicho de estimação?”, e você respondesse “é um animal!”, a pessoa voltaria a perguntar “Qual animal?”, no melhor dos casos. No pior, lhe chamaria de louco.

Então vamos resolver o problema do seu bicho de estimação da seguinte maneira:

type

```
TCachorro = class(TAnimal)
```

```
    procedure Latir;
```

```
end;
```

Agora sim, poderemos citar uma classe concreta. Até porque **existem** cachorros de verdade no nosso mundo, sob a forma real de cachorro. Animais também existem, porém, sob forma de cachorros, gatos, etc. Nunca como animais apenas.

Por vias formais, definimos como classe abstrata aquela que possui métodos abstratos. Métodos abstratos são aqueles que não possuem implementação física na classe. No Delphi eles são declarados por uma diretiva **abstract** depois da sua declaração no cabeçalho. Veremos isto mais adiante, numa próxima publicação. Mas ao contrário do Delphi, algumas linguagens como Java não permitem a instanciação de classes abstratas.

Se você tentar instanciar uma classe abstrata no Delphi, como TStrings por exemplo, ele lhe permitirá, embora lhe dê um aviso em compilação “*constructing instance of abstract class*”. O erro só ocorrerá em tempo de execução, quando você invocar um método abstrato. Daí, para evitar o problema, sempre instancie classes concretas, nunca as abstratas. Uma boa candidata a ser construída de maneira correta neste caso é a classe TStringList, descendente de TStrings.

Este tipo de permissividade do Delphi – de instanciar classes abstratas – leva ao questionamento de alguns pesquisadores, tanto em termos de consistência do projeto quanto da compreensão do modelo especificado. Java, nesse aspecto, parece ser mais fiel ao paradigma OO, por não permitir a compilação.

Construtores e Destruidores específicos

Suponha agora que aquele seu cachorro – aquele declarado na seção anterior – tenha um dono. Logo, nada nos custa dar um atributo *Dono* ao pobre animal, do tipo *string*. Suponha ainda que na regra do nosso projeto não possam existir cachorros sem dono, em nenhum momento, e nem se pode correr esse risco.

Se você considerar que usará o construtor padrão, derivado de TObject (aquele Create sem parâmetros), deverá admitir que o desenvolvedor que usar sua classe poderá esquecer de dar um dono ao cachorro. Isto, em alguns casos, poderá trazer conseqüências inesperadas ou até desagradáveis ao seu objeto.

Daí, em casos deste tipo utiliza-se um construtor específico à sua classe, declarando-o portador de parâmetros que sejam fundamentais à sua criação. Vejamos:

type

```
TCachorro = class(TAnimal)
  Dono: string;
  constructor Create(const Dono: string);
  procedure Latir;
end;
```

implementation

```
constructor TCachorro.Create(const Dono: string);  
begin  
  inherited Create;  
  Self.Dono := Dono;  
end;
```

A partir de agora, nenhum cachorro no modelo poderá ser instanciado sem que lhe haja o nome do dono. Nota-se que o nosso construtor faz uma chamada ao construtor da ancestral, através da palavra reservada **inherited** (que quer dizer *herdado*), que é quem de fato irá criar a classe na memória. Veremos as chamadas *inherited* em mais detalhes na próxima parte. A referência *Self*, vale informar, aponta para a instância em questão, fazendo diferenciar neste caso o que é o atributo *Dono* do parâmetro *Dono*.

Logo aproveitamos para falar que uma boa prática de desenvolvimento de classes, é sempre procurar obrigar a informação dos *dados importantes* da criação no seu construtor. O uso de construtores específicos se faz importante também quando se deseja executar alguma ação (código) no momento da criação do objeto.

De igual maneira, podemos personalizar o destruidor da nossa classe, pela declaração abaixo:

```
type  
TCachorro = class(TAnimal)  
  Dono: string;  
  constructor Create(const Dono: string);  
  destructor Destroy; override;  
  procedure Latir;  
end;
```

implementation

```
destructor TCachorro.Destroy;  
begin  
  ShowMessage('Eu precisava muito exibir esta mensagem antes de ser destruído!');  
  Inherited;  
end;
```

Esta declaração faz a nossa classe *sobrescrever* o destruidor padrão, chamando-o dentro do seu código. Observe que a chamada *inherited*, que é quem de fato irá destruir o objeto, vem por último. Na próxima parte, quando estivermos falando de polimorfismo, explicaremos melhor estes detalhes. A assinatura do destruidor é padrão, não podendo ser acrescida de parâmetros como fizemos com o Create.

Dica: Recomendamos aos desenvolvedores sempre evitem o uso explícito do *Destroy*. Ele não checa se a referência é nula antes de destruí-lo, o que pode ser desagradável em caso de ser nula. Procure sempre usar o método *Free* derivado de *TObject*, que faz recorrência ao *Destroy*, porém de maneira segura.

Parte III – Princípios: Polimorfismo

È mais um dos três requisitos fundamentais a uma linguagem ser considerada OO. Este recurso está diretamente ligado à herança, e a incrementa de tal maneira que os objetos que dele fazem uso adquirem um alto grau de flexibilidade e auto-suficiência. Nota-se que, pouco a pouco, os programadores que fazem uso deste recurso acabam por ter cada vez menos preocupação e responsabilidade sobre o comportamento de seus objetos, delegando-o para eles próprios.

O que é *polimorfismo*, afinal?

Bom, tentaremos explicar o que é polimorfismo em poucas linhas, muito embora isto não seja nada fácil. O polimorfismo, ao grosso modo, é a capacidade de objetos pertencentes a uma mesma classe se comportarem diferentemente, a depender da classe concreta a que eles pertencem.

A palavra “comportarem” certamente foi percebida na explicação acima, embora não tenha sido entendida direito. Comportamento, como já tratamos, tem a ver com métodos. E o polimorfismo se aplica exatamente a isso, aos métodos. Vejamos um exemplo de onde aplicá-lo:

```
unit Unit1;
```

```
type
```

```
  TVeiculo = class  
    Posicao: TPoint;  
    procedure MoverSe(const p: TPoint);  
end;
```

```
  TCarro = class(TVeiculo)  
    Pneus: Integer;  
end;
```

```
  TAviao = class(TVeiculo)  
    Asas: Integer;  
end;
```

No nosso exemplo temos três declarações de classes, sendo uma ancestral, TVeiculo, e duas descendentes, TCarro e TAviao. Nota-se que TVeiculo possui um método MoverSe, que por dedução, irá deslocar o nosso veículo em termos de posição. Nota-se ainda que ambos os subtipos de veículos – carro e avião – se movem.

Mas existe um problema no nosso modelo. Acontece que um avião se move de maneira diferente de um carro. Se estivéssemos falando em termos de animação gráfica, o nosso carro se moveria pela terra, girando os pneus. O avião, por sua vez, se moveria pelo ar, girando as turbinas.

Em outras palavras, a modelagem está correta em termos das classes. O método mover está relacionado com sua classe mais ancestral, o TVeiculo, e ele descreve muito bem **o que** deve ser feito. Mas não descreve com perfeição **como** isto deve ser feito. Aliás, este

é um grande propósito do polimorfismo em relação às classes: separar *o que* do *como*, ou os fins dos meios, como diria Maquiavel.

A proposta do polimorfismo aqui é permitir que o carro e o avião possam se mover de forma diferente e própria, sem afetar a declaração e a abstração da sua classe ancestral TVeiculo. Assim, será possível manter os dois tipos de objeto fiéis à declaração da classe dos veículos. Agora vamos corrigir nosso exemplo:

```
unit Unit1;
```

```
type
```

```
  TVeiculo = class  
    Posicao: TPoint;  
    procedure MoverSe(const p: TPoint); virtual;  
  end;
```

```
  TCarro = class(TVeiculo)  
    Pneus: Integer;  
    procedure MoverSe(const p: TPoint); override;  
  end;
```

```
  TAviao = class(TVeiculo)  
    Asas: Integer;  
    procedure MoverSe(const p: TPoint); override;  
  end;
```

E vamos implementá-lo:

```
implementation
```

```
procedure TVeiculo.MoverSe(const p: TPoint);  
begin  
  //aqui nada interessa ser feito  
end;
```

```
procedure TCarro.MoverSe(const p: TPoint);  
begin  
  ShowMessage('Estou girando minhas rodas sobre o solo');  
  Posicao := p;  
end;
```

```
procedure TAviao.MoverSe(const p: TPoint);  
begin  
  ShowMessage('Estou girando minhas turbinas no ar');  
  Posicao := p;  
end;
```

Agora vamos entender o que aconteceu. Primeiramente, declaramos o método na classe TVeiculo com uma diretiva **virtual**. Ainda não explicaremos o real significado disto neste parágrafo. Mas por enquanto vamos assumir que isto quer dizer que o método é *polimórfico*, ou seja, pode assumir outros comportamentos (diferentemente dos métodos estáticos). Depois declaramos os mesmos métodos nas classes descendentes com uma diretiva **override**, que é quem de fato irá implementá-los. E então, desenvolvemos os três métodos em código. O MoverSe de TVeiculo nada faz, e o MoverSe de cada tipo concreto de veículo o implementa cada um à sua maneira.

Pois eis que, caro quase-programador OO, nós demos implementações diferentes de um mesmo método para diferentes subclasses desta mesma classe. Traduzindo esta frase, o que fizemos foi dizer a TVeiculo que ele se move. Logo em seguida, demos a cada um dos subtipos de veículo o *direito* de se moverem cada um do seu jeito, e ainda sob a mesma chamada de método em TVeiculo.

Pois acontece que se referenciarmos uma instancia de TCarro por uma variável do tipo TVeiculo, e mandarmos que execute o método MoverSe, o nosso veículo **irá se mover como um carro**. E assim, chegamos ao grande pulo do gato da OO. Vejamos outro exemplo:

```
unit Unit1;
```

```
const
```

```
PI = 3.141592654;
```

```
RAIZ_DE_DOIS = 1.4142135;
```

```
type
```

```
TFigura = class
```

```
Centro: TPoint;
```

```
constructor Create(const Pos: TPoint);
```

```
procedure Desenhar; virtual;
```

```
function Area: Real; virtual;
```

```
end;
```

```
TCirculo = class(TFigura)
```

```
Raio: Integer;
```

```
procedure Desenhar; override;
```

```
function Area: Real; override;
```

```
end;
```

```
TQuadrado = class(TFigura)
```

```
Lado: Integer;
```

```
procedure Desenhar; override;
```

```
function Area: Real; override;
```

```
end;
```

implementation

```
constructor TFigura.Create(const Pos: TPoint);
```

```
begin
```

```
  inherited Create;
```

```
  Centro := Pos;
```

```
end;
```

```
procedure TFigura.Desenhar;
```

```
begin
```

```
  //nada aqui
```

```
end;
```

```
function TFigura.Area: Real;
```

```
begin
```

```
  //nem aqui
```

```
end;
```

```
procedure TCirculo.Desenhar;
```

```
var Rect: TRect;
```

```
begin
```

```
  Rect.Left := Centro.X - Round(Raio/RAIZ_DE_DOIS);
```

```
  Rect.Top := Centro.Y - Round(Raio/RAIZ_DE_DOIS);
```

```
  Rect.Right := Centro.X + Round(Raio/RAIZ_DE_DOIS);
```

```
  Rect.Bottom := Centro.Y + Round(Raio/RAIZ_DE_DOIS);
```

```
  Form1.Canvas.Ellipse(Rect);
```

```
end;
```

```
function TCirculo.Area: Real;
```

```
begin
```

```
  Result := 2*PI*Raio;
```

```
end;
```

```
procedure TQuadrado.Desenhar;
```

```
var Rect: TRect;
```

```
begin
```

```
  Rect.Left := Centro.X - Lado div 2;
```

```
  Rect.Top := Centro.Y - Lado div 2;
```

```
  Rect.Right := Centro.X + Lado div 2;
```

```
  Rect.Bottom := Centro.Y + Lado div 2;
```

```
  Form1.Canvas.Rectang(Rect);
```

```
end;
```

```
function TQuadrado.Area: Real;
```

```
begin
```

```
  Result := Power(Lado, 2);
```

```
end;
```

Bom, este nosso último exemplo talvez tenha trazido a coisa um pouco mais à prática, até porque imaginamos que o leitor deve ter pensado por muitas vezes “mas para que eu vou querer animais, vegetais ou veículos dentro do meu programa?”. Este nosso exemplo traz à tona o conceito de polimorfismo e tanto pode como deve ser implementado para que o leitor o veja acontecer.

Como se pode notar, todas as chamadas à execução fazem referência a um objeto Form1. Obviamente estamos nos referindo ao *form* principal do nosso exemplo. Entretanto isto não é uma boa prática de programação, levando a uma situação que os analistas e engenheiros de *software* chamam de *forte acoplamento*.

Forte acoplamento quer dizer que as suas classes são muito dependentes deste *form*, e uma mudança nesta parte pode requerer uma mudança em todas as classes fortemente acopladas. Isto também compromete a reutilização de suas classes para outros projetos. Mas não vamos nos preocupar com isso aqui. Em próximas publicações iremos abordar o acoplamento e boas práticas de programação.

O código abaixo é uma sugestão de aplicação do código exemplificado acima, para que se veja o polimorfismo em funcionamento:

type

```
TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
private
  { Private declarations }
public
  Figuras: array [1..10] of TFigura;
  { Public declarations }
end;
```

implementation

```
procedure TForm1.FormCreate(Sender: TObject);
var i: Integer; NovaPos: TPoint;
begin
  Randomize;
  for i := 1 to 10 do
  begin
    NovaPos.X := Random(Self.Width);
    NovaPos.Y := Random(Self.Height); //sorteia a posição na janela
    if Random(10) > 5 then //sorteia qual figura irá instanciar
      Figuras[i] := TCirculo.Create(NovaPos)
    else
      Figuras[i] := TQuadrado.Create(NovaPos)
    end;
  end;
end;
```

```

procedure TForm1.Button1Click(Sender: TObject);
var i: Integer;
begin
    Self.Repaint;
    for i := 1 to 10 do
        Figuras[i].Desenhar;
end;

```

Note que, ao executar Button1Click(), todas as figuras serão desenhadas, cada uma a seu modo.

Métodos Abstratos

Vimos que o que torna uma classe abstrata de fato é a existência de métodos abstratos. Pois bem que métodos abstratos são aqueles que *não possuem* nenhuma implementação naquela classe, deixando para serem implementados apenas pelas classes descendentes.

Voltemos ao nosso último exemplo, envolvendo círculos e quadrados. Note que o método Desenhar foi declarado virtual na classe TFigura, e implementado nas classes TCirculo e TQuadrado. Entretanto, este método ainda é implementado nesta classe, pois não foi declarado abstrato. Existe uma declaração vazia deste método em TFigura, por mais que nenhum código tenha sido colocado ali dentro.

Acontece que se fosse instanciado, juntamente com os círculos e quadrados, um objeto concreto do tipo TFigura, no momento em que o método Desenhar fosse invocado, ele seria executado de fato. Para ter certeza disto, implemente-o da seguinte maneira:

```

procedure TFigura.Desenhar;
begin
    ShowMessage('Erro: este método não deveria ser chamado!!!');
end;

```

E procure instanciar também objetos do tipo TFigura, usando o construtor TFigura.Create dentro do *for-loop* no método FormCreate. Você perceberá que este método será executado fiel como foi declarado na classe TFigura. Porém, se quisermos declarar o método Desenhar como abstrato, removeremos a sua implementação e faremos a sua declaração da seguinte maneira:

```

type
    TFigura = class
        Centro: TPoint;
        constructor Create(const Pos: TPoint);
        procedure Desenhar; virtual; abstract;
        function Area: Real; virtual;
    end;

```

A razão da existência de métodos abstratos é “obrigar” que o desenvolvedor sempre implemente estes métodos nas classes inferiores, estabelecendo uma boa regra de consistência no comportamento das classes.

Mas porque então não declaramos logo todos os métodos polimórficos como abstratos? Acontece que um método polimórfico não deve ser necessariamente abstrato. A classe pode ser abstrata, contendo métodos abstratos, e ainda assim conter um método virtual não-abstrato entre eles. Dessa maneira, podemos estabelecer métodos polimórficos com um comportamento *default* para as outras classes, e a menos que a classe descendente o sobrescreva, ela se comportará desta maneira.

Em contrapartida, o uso de métodos abstratos pode ser desagradável caso o leitor realmente queira instanciar aquela classe. Aí, novamente, cairemos na situação em que a classe **não deveria** ser definida abstrata, e a chamada deste método irá retornar um erro em tempo de compilação com a mensagem “*abstract error*”. Como já lamentamos na parte anterior, a permissividade de se construir classes abstratas é particular do Delphi.

Acessando os métodos originais

Suponhamos que você está escrevendo uma classe descendente de outra, aplicando polimorfismo em alguns de seus métodos. Suponha ainda que seu método não foi declarado abstrato, e ele possui uma implementação. Vejamos o exemplo:

```
TCirculoPartido = class(TCirculo)
  procedure Desenhar; override;
end;
```

Esta nova classe que inventamos precisa desenhar um círculo com duas linhas de diâmetro, partindo-o em forma de cruz. Sabe-se que, além de ser considerada um círculo, ela também precisa de muitas das implementações já feitas em **TCirculo**, modificando apenas o método **Desenhar**, e por isso fica óbvia uma situação de herança. Ora, se vamos desenhar o círculo e seus dois diâmetros, eis que faremos a mesma coisa que **TCirculo.Desenhar** já fazia, com algo a mais. Para isso, existe um meio de evitar que tenhamos que reescrever todo este código do círculo dentro de **TCirculoPartido** novamente. Assim, chamamos a palavra reservada **inherited** com o nome do método. Vejamos como fica o nosso novo código:

```
procedure TCirculoPartido.Desenhar;
var Origem, Destino: TPoint;
  procedure DesenhaLinha;
  begin
    Form1.Canvas.MoveTo(Origem.X, Origem.Y);
    Form1.Canvas.LineTo(Destino.X, Destino.Y);
  end;
begin
  inherited Desenhar; //que surte o mesmo efeito de chamar somente inherited;
```

```
Origem.Y := Centro.Y;
Origem.X := Centro.X - Raio;
Destino.Y := Centro.Y;
Destino.X := Centro.X + Raio;
DesenhaLinha;
Origem.X := Centro.X;
Origem.Y := Centro.Y - Raio;
```

```
Destino.X := Centro.X;  
Destino.Y := Centro.Y + Raio;  
DesenhaLinha;  
end;
```

O objetivo do *inherited* é informar ao compilador que o método chamado **deve** ser o da classe imediatamente superior (classe pai), ignorando um mesmo método declarado na classe atual. Em Java equivale a invocar *super*.

Métodos virtuais versus Métodos dinâmicos

Bom, fugindo um pouco da linha do assunto, há alguns parágrafos acima tivemos que usar uma palavra reservada **virtual** como diretiva depois de um método, para *dizer* ao compilador que ele era polimórfico. Eis que agora explicaremos o porquê.

Existem dois tipos de métodos polimórficos, quanto à implementação interna do compilador: métodos virtuais (virtual) e métodos dinâmicos (dynamic). À visão do programador, eles são idênticos, e fazem exatamente a mesma coisa. A diferença está em *como* o compilador recorre a estes métodos.

Sabe-se que métodos virtuais contra métodos dinâmicos são uma razão entre eficiência contra tamanho do código gerado. Muito embora você perceba chamadas à diretiva **dynamic** nos componentes da VCL, a própria Borland recomenda que se use o virtual na grande maioria dos casos. Para saber mais detalhes sobre estes dois tipos de implementação, recomendamos que consulte a documentação da Borland, disponível na ajuda do Delphi, ou no site <http://www.borland.com>.

Parte IV – Princípios: Encapsulamento

O encapsulamento em resumo é a maneira de fazer com que um objeto exponha apenas o que *é* necessário ser exposto, mantendo como próprio e privado tudo aquilo que não seja de interesse de outros objetos que o vejam por fora. Desta maneira poderemos separar “os dois mundos”, a que nos referimos no título – o mundo exterior ao objeto e seu mundo privado.

Mantendo uma vida particular

Imagine um aparelho de TV. Ele normalmente é constituído de um painel, um visor, e alguns circuitos internos. O painel é o seu controle, acessível ao seu usuário através de alguns botões. De igual maneira, o visor procura exibir, diretamente para o telespectador, as imagens que lhe são requeridas. Estas duas partes são obviamente necessárias ao acesso manual, satisfazendo muito bem o seu propósito.

Entretanto, determinados circuitos e controles não estão disponíveis no painel ou na caixa deste equipamento. Eles estão isolados *dentro* do aparelho, a fim de evitar o acesso manual, que poderia ser perigoso tanto para quem tentasse acessá-lo, quanto poderia comprometer o bom funcionamento do aparelho de TV.

Por isso é importante que os objetos tenham alguns de seus elementos privados, dentre aqueles que são declarados nos seus atributos e métodos. Com esta privacidade será possível manter a compreensão, com um código mais limpo e mais legível. Além disso, o acesso a determinados atributos ou métodos perigosos dos objetos será restrito.

O objetivo do encapsulamento é tratar o objeto como uma **caixa preta**, que faz certas coisas, e disponibiliza para que os seus programadores – que neste caso, devem ser tratados mais como “usuários” – saibam *o que* ele faz. *Como* o objeto faz, não seria de todo interessante saber. E isto é um ponto fundamental para se reduzir a complexidade do sistema à visão do programador.

Pois eis que o exemplo do aparelho de TV é uma boa analogia quanto ao encapsulamento dos objetos. Além do acesso total não ser muito conveniente, os usuários não costumam estar devidamente qualificados para acessar as centenas de circuitos dentro do aparelho. Da mesma forma, os outros objetos de um programa certamente não estão qualificados ao acesso total de todos os objetos.

Classificando os membros por visibilidade

Os membros de uma classe são classificados por três escopos principais de visibilidade: **públicos** (public), **privados** (private) e **protegidos** (protected). No Delphi especificamente surge um tipo chamado *published*, que é uma variante do público, mas que não está relacionado especificamente com a OO, e sim com os componentes da VCL. Não trataremos de *published*.

Os atributos e métodos públicos são aqueles que estão disponíveis para tudo e todos acessarem, desde que tenham visibilidade do objeto. Eles são visíveis tanto externa quanto internamente ao objeto, e visam disponibilizar a comunicação do objeto com os

demais elementos do modelo. Membros são públicos quando declarados numa seção **public** da classe a que pertence.

Os membros privados por sua vez são aqueles que *somente* o objeto deve saber, não devendo estar disponíveis para os objetos externos. Eles devem ser declarados na seção **private** da classe. As seções podem ser declaradas em qualquer ordem, desde que dispostas sempre antes dos membros relativos à sua declaração. Vejamos um exemplo:

type

```
TCarro = class
private
  FMotor: TMotor;
  procedure AlimentarBombaDeCombustivel;
  procedure LiberarValvulaDeInjecao;
  procedure InjetarCombustivel;
  procedure AcionarPastilhas;
  procedure DispararArranque;
public
  Pneus: array [1..5] of TPneu;
  procedure Ligar;
  procedure Acelerar;
  procedure Freiar;
end;
```

implementation

```
procedure TCarro.Ligar;
begin
  DispararArranque;
  InjetarCombustivel;
end;
```

```
procedure TCarro.Acelerar;
begin
  InjetarCombustivel;
end;
```

```
procedure TCarro.Freiar;
begin
  AcionarPastilhas;
end;
```

```
procedure TCarro.AlimentarBombaDeCombustivel;
begin
  {aqui deverá ser implementado o código referente à alimentação da bomba de
  combustível}
end;
```

```
procedure TCarro.InjetarCombustivel;
begin
```

```
AlimentarBombaDeCombustivel;  
LiberarValvulaDeInjecao;  
end;
```

```
procedure TCarro.AcionarPastilhas;  
begin  
  {aqui deverá ser implementado o código referente ao acionamento das pastilhas  
  de freio}  
end;
```

```
procedure TCarro.DispararArranque;  
begin  
  {aqui deverá ser implementado o código referente ao disparo do motor de  
  arranque}  
end;
```

O que fizemos no exemplo? Bom, declaramos alguns métodos e atributos privados e outros públicos, de modo que o que está privado – dentro da cláusula *private* – não estará acessível para uso por qualquer outro elemento do sistema, senão dentro do próprio código da classe – diferentemente do que está declarado no *public*, que pode ser usado por qualquer um que consiga enxergar o objeto. Vale ressaltar que por convenção declaramos os atributos privados começando-os pela letra “F” – do inglês *Field* (campo).

Note que existe uma razão para isso. Todos os métodos declarados públicos estão mais relacionados a funcionalidades externas do objeto, ou seja, algo que o usuário deve saber: “o que” deve ser feito. Lá estão os métodos *Ligar*, *Freiar* e *Acelerar* do carro. Qual motorista não deveria fazer estas três coisas?

Em contrapartida, nada do que foi declarado em *private* deve ou precisa ser visto externamente ao carro. Somente ele deve saber que precisa injetar combustível ao acelerar, ou acionar as pastilhas de freio ao se freiar. Isto é “como” deve ser feito, e não há por que isto interessar ao mundo de fora.

Um terceiro tipo de visibilidade dos membros de uma classe é o *protegido*. Os membros protegidos, declarados na seção **protected**, são privados ao objeto tanto quanto os declarados na seção *private*, exceto por um detalhe. Eles são visíveis a todas as classes descendentes, como se fossem membros privados delas.

Assim, seria interessante que instâncias das classes *TCarroPasseio* e *TCarroEsporte*, descendentes de *TCarro*, pudessem acessar alguns de seus elementos privados, uma vez que elas também pertencem a esta classe. Agora vamos reformular o cabeçalho da nossa classe:

```
type  
  TCarro = class  
  private  
    FMotor: TMotor;  
  procedure AlimentarBombaDeCombustivel;  
  procedure LiberarValvulaDeInjecao;
```

```

procedure AcionarPastilhas;
protected
procedure InjetarCombustivel;
procedure DispararArranque;
public
  Pneus: array [1..5] of TPneu;
procedure Ligar;
procedure Acelerar;
procedure Freiar;
end;

```

A razão desta mudança é que o desenvolvedor desta classe imaginou o quanto seria útil disponibilizar os métodos *InjetarCombustivel* e *DispararArranque* para as classes descendentes dela, e o fez ao declará-los como protegidos. Note que os métodos declarados em *private* realmente não precisariam serem disponibilizados, a não ser que isto fosse julgado importante.

Propriedades: atributos a uma visão externa

A orientação a objetos diz, em seu conceito geral, que todo e qualquer atributo deve ser acessado apenas através de métodos públicos. Daí, dispor atributos publicamente não seria uma boa prática de programação.

O objetivo desta restrição é garantir a consistência do estado do objeto. Convenhamos que qualquer objeto com um bom encapsulamento saberá, através destes métodos, exatamente a melhor forma de manter a leitura ou escrita de seus dados. Estes métodos - popularmente conhecidos como *gets* e *sets* - pertencem a uma filosofia bem respeitada por programadores Java em geral.

A Borland, por sua vez, adotou uma medida alternativa e interessante de promover a segurança de seus atributos: o uso de *propriedades*. Esta medida se trata, na verdade, de encapsular os atributos sob métodos *gets* e *sets*. Isto dará ao desenvolvedor-usuário desta classe a “sensação” de que ele está manipulando os atributos internos do objeto diretamente, sem que de fato esteja. Vejamos:

```

type
  TMinhaClasse = class
private
  FAtributo: Integer;
  FInicializado: Boolean;
procedure SetAtributo(const Value: Integer);
function GetAtributo: Integer;
public
property Atributo: Integer read GetAtributo write SetAtributo;
end;

```

```

var
  MeuObjeto: TMinhaClasse;

```

implementation

```
procedure TMinhaClasse.SetAtributo(const Value: Integer);  
begin  
  if (Value < 0) or (Value > 100) then  
    raise ELimiteException.Create('Este valor está fora dos limites' +  
      ' estabelecidos para este objeto');  
  FAtributo := Value;  
end;
```

```
function TMinhaClasse.GetAtributo: Integer;  
begin  
  if not FInicializado then  
    raise EEstadoException.Create('Este objeto ainda não' +  
      ' foi inicializado');  
  Result := FAtributo;  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  if MeuObjeto.Atributo <= 20 then  
    begin  
      ShowMessage('Este valor está muito baixo. Vamos aumentá-lo.');
```

```
      MeuObjeto.Atributo := 35;  
    end;  
end;
```

No código acima, note que declaramos um atributo privado qualquer, *FAtributo*. Este atributo é algo de *íntimo do objeto*, e deve respeitar a algumas regras internas da classe, que nem todos os que acessam a classe necessariamente devem saber. Mas querendo ou não, terão que respeitar. Propriedades são apenas uma forma de abstração do código, e o compilador, na realidade, entende como se você estivesse chamando os métodos *GetAtributo* e *SetAtributo* diretamente.

Nota-se que nossa classe garante a integridade de leitura e escrita do atributo *FAtributo*, através do uso de exceções. O acesso dele está sendo feito pela propriedade *Atributo*, que é entendida pelo compilador como o acesso aos métodos *GetAtributo* e *SetAtributo*. No primeiro, garante-se que nenhuma instância desta classe terá *FAtributo* lido, sem que *FInicializado* esteja com o valor *True*. No segundo, garante-se que *nunca* lhe será atribuído um valor fora do limite de 0 a 100.

Dica: A IDE do Delphi possui um atalho para automaticamente criar os métodos *Gets* e *Sets*, declarando-os na propriedade, e criando o seu corpo automaticamente na seção *implementation*. Para isto, basta declarar a propriedade com o tipo, e pressionar [*control* + *shift* + *C*].

É de se observar ainda que as propriedades não necessariamente precisam representar um atributo-imagem pelo lado de fora, uma vez que se tratam puramente de métodos de entrada e saída. Elas podem ser apenas uma *abstração* de atributos, que podem ser

físicos ou não. O importante é que o programador-usuário entenda que está acessando uma propriedade do objeto. Assim criaremos propriedade *virtual*, e o programador usuário da classe simplesmente acredita que está acionando um atributo. Exemplo:

type

```
TDatabase = class
private
  procedure SetActive(const Value: Boolean);
  function GetActive: Boolean;
protected
  procedure Connect;
  procedure Disconnect;
  function IsConnected: Boolean;
public
  property Active: Boolean read GetActive write SetActive;
end;
```

implementation

```
procedure TDatabase.SetActive(const Value: Boolean);
begin
  if Value then
    Connect
  else
    Disconnect;
end;

function TDatabase.GetActive: Boolean;
begin
  Result := IsConnected;
end;
```

A sintaxe da declaração dos membros de leitura e escrita nas propriedades admite tanto métodos quanto atributos de mesmo tipo da propriedade. Não serão estudados mais detalhes sobre propriedades aqui, porque elas não pertencem à filosofia do paradigma OO. Para ver este assunto mais detalhadamente, nós recomendamos consultar a ajuda do Delphi.

Sobrecarga de Métodos

Um outro conceito relevante quanto à visibilidade de métodos é a *sobrecarga*. Sobrecarga é a declaração de dois métodos diferentes numa mesma classe, com o mesmo nome. O que deve diferir os métodos são os seus parâmetros.

Se existe uma funcionalidade na sua classe que executa coisas diferentes em dois métodos com parâmetros diferentes, mas que tenham um mesmo propósito, então aparentemente não há impecílios para que se dê o mesmo nome aos métodos. Basta sobrecarregá-los. Para isso, usa-se a diretiva **overload**. A sobrecarga pode inclusive facilitar a vida de quem irá usar a sua classe. Vejamos um exemplo:

type

TColecao = **class**

public

procedure Adiciona(const Item: TItem); **overload;**

procedure Adiciona(const Colecao: TColecao); **overload;**

end;

A partir do código de exemplo acima, podemos usar a chamada de método *Adiciona* tanto para adicionar itens quanto para adicionar Coleções. Nota-se que o procedimento executado será diferente em cada um dos métodos. A maneira com que se adiciona um *TItem* é diferente de como se adiciona um *TColecao*. Mas, como mais um caso de fins *versus* meios, o propósito é o mesmo.

Visibilidade entre classes distintas

A rigor, nenhuma classe enxerga os atributos privados de outra. Assim diz a regra da boa vizinhança entre as classes, na OO. Sabe-se que o ideal é que cada classe conheça o *menos possível* dos elementos internos da outra na OO, mas existe um *porém*. Um não, dois, pois existem situações em que pode ser necessário se estender a visibilidade entre as classes.

O primeiro caso refere-se aos membros protegidos. Ora convenhamos que se um método é declarado protegido, isto significa que, muito embora não seja desejável que ele seja visto por fora, é interessante que as subclasses façam uso dele. Portanto, o método `Tcontrol.Click`, embora não seja público, é necessário que seja acessível tanto por um `TEdit`, quanto por um `TButton` ou por qualquer subclasse descendente de `TControl`.

O segundo porém tem a ver com *classes amigas*. A “amizade” entre as classes é algo conhecido no `Ansi C++`. Ela se caracteriza por estabelecer, através da palavra reservada *Friendly*, uma espécie de relacionamento entre duas ou mais classes, de modo que garanta a visibilidade de algumas de suas partes privadas entre elas.

No Delphi, esta cumplicidade entre as classes não é declarada por meio de palavras reservadas, mas ela existe. Duas classes passam a ter visibilidade mútua de seus atributos privados quando são declaradas na mesma *unit*. Daí, se for interessante compartilhar atributos privados entre duas ou mais classes, as declare sempre dentro de um mesmo arquivo “.pas”.

Entretanto, o acúmulo de classes diversas numa mesma unidade de código pode acabar comprometendo a segurança de seus objetos. Daí uma boa prática de programação que propõe dar ênfase ao encapsulamento é procurar declarar as classes no máximo de arquivos fontes separados o possível. Se no seu modelo não existe um motivo bem forte para que uma classe enxergue os elementos privados de outra, então as separe.

Público, privado ou protegido: quando devo declarar?

O discernimento do grau de visibilidade é algo trivial para um programador OO experiente. Mas ainda não é para quem não está habituado com o encapsulamento. O critério de quando declarar um atributo como público ou como privado é adquirido com a prática. Mas sempre sugerimos tentar discernir, dentre as funcionalidades e propriedades dos objetos, o que pode ser separado entre *o que fazer* dos detalhes de *como se fazer*. Os primeiros são forte candidatos a serem privados ou protegidos.

Um critério alternativo adotado por alguns desenvolvedores para definir o encapsulamento dos objetos, segue a frase “*tudo é privado até que se prove o contrário*”. Assim, declaram-se todos os métodos como privados, até que uma subclasse realmente precise acessá-los, para que se tornem *protegidos* – ou até que realmente seja preciso acessá-los publicamente, e se tornarem públicos. Esta filosofia, apesar de ser muito questionada, garante um bom nível de segurança do objeto.

Quanto aos atributos, recomendamos fortemente que nunca sejam declarados públicos, sempre procurando usar propriedades para a sua leitura e escrita externas. Esta é uma boa prática de OO.