Conceitos de Programação Orientada a Objetos

Antes de partir para a linguagem propriamente dita, vamos revisar alguns conceitos básicos de Programação Orientada a Objetos.

Classe: Definição de tipo dos objetos, modelo de objeto.
Objeto: Instância de classe, variável cujo tipo é uma classe.
Atributos: Variáveis de instância, são os dados de um objeto.
Métodos: Funções e procedimentos de um objeto.
Propriedades: *Apelido* usado para evitar o acesso direto aos atributos de um objeto, onde podemos especificar métodos que serão usados para ler e atribuir seus valores a esses atributos.
Mensagens: Chamada de métodos, leitura e atribuição de propriedades.
Encapsulamento: Conjunto de técnicas usadas para limitar o acesso aos atributos e métodos internos de um objeto.
Herança: Possibilidade de criar uma classe descendente de outra, aproveitando seus métodos, atributos e propriedades.
Ancestral: Super classe ou classe de base, a partir da qual outras classes podem ser criadas.
Descendente: Subclasse.
Hierarquia de Classes: Conjunto de classes ancestrais e descendentes, geralmente representadas em uma árvore hierárquica.
Polimorfismo: Capacidade de redefinir métodos e propriedades de uma classe em seus descendentes.

Estrutura de Units
Vamos examinar o código gerado para um novo Form, identificando as principais seções de uma Unit típica. Crie uma nova aplicação e observe na Unit principal as seguintes cláusulas.

Unit: A primeira declaração de uma unit é seu identificador, que é igual ao nome do arquivo.
Interface: Seção interface, onde ficam declarações que podem ser usadas por outras Units.
Uses: Na cláusula uses fica a Lista de Units usadas.
Type: Na cláusula type fica a definição de tipos, aqui temos a declaração da classe do Form.
Var: Na cláusula var são declaradas as variáveis, aqui temos a declaração da instância do Form.
Implementation: Na seção implementation ficam as definições dos métodos.
End: Toda Unit termina com um *end* a partir de onde qualquer texto é ignorado.

Variáveis
No Delphi, toda variável tem que ser declarada antes de ser utilizada. As declarações podem ser feitas após a palavra reservada *var*, onde são indicados o nome e o tipo da variável. Os nomes de variáveis não podem ter acentos, espaços ou caracteres especiais como &, $ ou % e o primeiro caractere de um nome de variável tem que ser uma letra ou um sublinhado. O Delphi ignora o caso das letras.

Variáveis Globais
As variáveis abaixo são globais, declaradas da *Interface* da Unit. Podem ser acessadas por qualquer Unit usuária.

```
var
  I: Integer;
  Usuario: string;
  A, B, Soma: Double;
  Ok: Boolean;
```

Variáveis Locais
As variáveis abaixo são locais ao método, ou seja elas só existem dentro do método, não podem ser acessadas de fora, mesmo que seja na mesma Unit. Na verdade essas variáveis são criadas quando o método é chamado e destruídas quando ele é encerrado, seu valor não é persistente.

```
procedure TFrmExemplo.BtnTrocarClick(Sender: TObject);
var
  Aux: string;
begin
  Aux :=  EdtA.Text;
  EdtA.Text := EdtB.Text;
  EdtB.Text := Aux;
end;
```

Atributos

Os atributos são variáveis de instância. Para declarar um atributo em uma classe basta definir o identificador e o tipo do atributo na declaração da classe, feita na seção *type* da *Interface* da Unit, como abaixo.

```
type
  TFrmSomar = class(TForm)
  private
    { Private declarations }
    A, B: Double;
  public
    { Public declarations }
    Soma: Double;
  end;
```

Encapsulamento

Os principais níveis de visibilidade dos atributos e métodos de uma classe são mostrados abaixo.

Nível	**Visibilidade**
Private	Os itens declarados nesse nível só podem ser acessados na mesma unit.
Public	Nesse nível, qualquer unit usuária pode acessar o item.
Protected	Os itens só poderam ser acessados em outra unit se for uma classe descendente.
Published	É o nível default, igual ao public, mas define propriedades e eventos usados em tempo de projeto.

Classes

Classes são tipos de objetos, uma classe é declarada na cláusula type da seção interface e os métodos são definidos na seção implementation. Examine o código de um Form para identificar os elementos de sua classe.

```
interface
type
  TFrmSomar = class(TForm)
    EdtA: TEdit;
    EdtB: TEdit;
    BtnSoma: TButton;
    procedure BtnSomaClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

implementation

procedure TFrmSoma.BtnSomaClick(Sender: TObject);
begin
  ShowMessage(EdtA.Text + EditB.Text);
end;
```

Objetos

Um Objeto é tratado como uma variável cujo tipo é uma classe. A declaração de objetos é igual à declaração de uma variável simples, tendo no lugar do tipo a classe do objeto.

```
var
  FrmSomar: TFrmSomar;
```

Literais

Valores literais são valores usados em atribuições e expressões. Cada tipo tem uma sintaxe diferente.

Tipo	Definição
Inteiro	Sequência de dígitos decimais (0 a 9), sinalizados ou não
Inteiro Hexadecimal	Sequência de digitos hexadeciamis (0 a F), precedidos por um $
Real	igual ao tipo inteiro, mas pode usar separador decimal e notação científica
Caractere	Letra entre apostrofosou o caractere # seguido de um número inteiro entre 0 e 255.
String	Sequência de caracteres delimitados por apóstrofos

Constantes

São declaradas na seção *const*, podem ser usadas como variáveis, mas não podem ser alteradas. Geralmente o nome das constantes é escrito em letras maiúsculas e na declaração dessas constantes não é indicado o tipo.

```
const
  G = 3.94851265E-19;
  NUM_CHARS = '0123456789';
  CR = #13;
  SPACE = ' ';
  MAX_VALUE = $FFFFFFFF;
```

Constantes Tipadas

Na verdade, constantes tipadas são variáveis inicializadas com valor persistente, que podem ser alteradas normalmente, como qualquer variável. A única diferença de sintaxe entre constantes tipadas e simples é que o tipo da constante é indicado explicitamente na declaração. Se uma constante tipada for declarada localmente, ela não será destruída quando o método for encerrado. Para diferenciar das constantes normais, costuma-se declarar estas com letras de caso variável, como abaixo.

```
const
  Cont: Integer = 1;
  Peso: Double = 50.5;
  Empresa: string = 'SENAC';
```

Instruções

Os programas são compostos por instruções, que são linhas de código executável. Exemplos de instruções simples são atribuições, mensagens entre objetos, chamadas de procedimentos, funções e métodos, como mostradas abaixo. As instruções podem ser divididas em várias linhas, o que indica o fim de uma instrução é o ponto e vírgula no final. Quando uma instrução é quebrada, costuma-se dar dois espaços antes das próximas linhas, para melhorar a leitura do código.

```
Caption := 'Gabba Gabba Hey!';
Form2.ShowModal;
Application.MessageBox('Você executou uma operação ilegal, o programa será finalizado.',
  'Falha geral', MB_ICONERROR);
```

Você pode usar várias instruções agrupadas em uma instrução composta, como se fosse uma só instrução. Uma instrução composta delimitada pelas palavras reservadas *begin* e *end*. Toda instrução, simples ou composta, é terminada com um ponto-e-vírgula.

```
if CheckBox1.Checked then
begin
  ShowMessage('O CheckBox será desmarcado.');
  CheckBox1.Checked := False;
end;
```

Estilo de Codificação
As instruções e todo o código de uma Unit devem ser distribuídos para facilitar o máximo a leitura. Para isso, podemos usar a indentação, geralmente de dois espaços para indicar os níveis de código. Procure criar um estilo próprio, que melhor se molde à sua realidade. Se for desenvolver em grupo, é melhor que todos usem o mesmo estilo para evitar confusões.
Comentários
Existem 3 estilos de comentário no Delphi, como mostrado abaixo.
(* Comentário do Pascal Padrão *)
{ Comentário do Turbo Pascal }
// Comentário de linha do C++
Cuidado com as diretivas de compilação, pois elas são delimitadas por chaves e podem ser confundidas com comentários. A diretiva de compilação mostrada abaixo é incluída em todas as Units de Forms.
{$R*.DFM}
Tipos de Dados Padrão
O Delphi trata vários tipos de dados padrão, segue uma descrição sucinta desses tipos.

Tipos Inteiros
São tipos numéricos exatos, sem casas decimais. O tipo *Integer* é o tipo inteiro padrão.

Tipo	Tamanho em Bytes	Valor mínimo	Valor máximo
ShortInt	1	-128	127
SmallInt	2	-32767	32767
LongInt	4	-2147483648	2147483647
Byte	1	0	255
Word	2	0	65535
Integer	4	-2147483648	2147483647
Cardinal	4	0	2147483647

Tipos Reais
São tipos numéricos com casas decimais. O tipo *Double* é o tipo real padrão.

Tipo	Tamanho em Bytes	Valor Mínimo	Valor Máximo	Dígitos Significativos
Real	6	10^{-39}	10^{38}	11-12
Single	4	10^{-45}	10^{38}	7-8
Double	8	10^{-324}	10^{308}	15-16
Extended	10	10^{-4932}	10^{4392}	19-20
Comp	8	-10^{18}	10^{18}	19-20
Currency	8	-10^{12}	10^{12}	19-20

Tipos Texto
Os tipos texto podem operar com caracteres simples ou grupos de caracteres. O tipo texto padrão é o tipo string.

Tipo	Descrição
Char	Um único caractere ASCII
String	Texto alocado dinamicamente, pode ser limitado a 255 caracteres conforme configuração
PChar	String terminada em nulo (#0), usada geralmente nas funções da API do Windows

O operador + pode ser usado para concatenar strings e você pode usar uma variável do tipo string como uma lista de caracteres.
ShowMessage('5ª letra do título da janela: ' + Caption[5]);
Label1.Text := '2ª letra do Edit: ' + Edit1.Text[2];
Existem várias funções de manipulação de strings, veja algumas das mais importantes mostradas abaixo.

Função	Descrição
AnsiCompareText	Compara 2 strings sem sensitividade de maiúsculas/minúsculas
AnsiLowerCase	Converte todas as letras de uma string para minúsculas
AnsiUpperCase	Converte todas as letras de uma string para maiúsculas
Copy	Retorna parte de uma string
Delete	Apaga parte de uma string
Insert	Insere uma string em outra
Length	Número de caracteres de uma string
Pos	Posição de uma string em outra
Trim	Remove todos os espaços de uma string
TrimLeft	Remove os espaços à esquerda de uma string
TrimRight	Remove os espaços à direita de uma string
Format	Formata uma string com uma série de argumentos de vários tipos

Por exemplo, para comparar o texto de dois Edits, poderíamos usar a função AnsiCompareText.

```
if AnsiCompareText(EdtA.Text, EdtB.Text) = 0 then
  ShowMessage('O texto dos dois Edits são iguais.');
```

A função Format é especialmente útil na formatação de strings, veja alguns exemplos.

```
ShowMessage(Format('O número %d é a parte inteira do número %f.', [10, 10.5]));
ShowMessage(Format('Este texto%sfoi formatado%susando o caractere #%d.', [#13, #13, 13]));
ShowMessage(Format('O preço do livro %s é %m.', ['Como Programar em Delphi', 50.7]));
```

Um detalhe que deve ser observado é que as propriedades dos objetos não podem ser usadas como variáveis em funções. Veja a declaração do procedimento Delete no help.

```
procedure Delete(var S: string; Index, Count:Integer);
```

Digamos que você deseje apagar as 5 primeiras letras de um Edit, como a string do Delete é variável, não poderia usar o código abaixo.

```
Delete(Edit1.Text, 1, 5);
```

Para você poder fazer a operação desejada, teria que usar uma variável como variável auxiliar.

```
var
  S: string;
begin
  S := Edit1.Text;
  Delete(S, 1, 5);
  Edit1.Text := S;
end;
```

Tipos Ordinais

Tipos ordinais são tipos que tem uma seqüência incremental, ou seja, você sempre pode dizer qual o próximo valor ou qual o valor anterior a um determinado valor desses tipos. São tipos ordinais o Char, os tipos inteiros, o Boolean e os tipos enumerados. Algumas rotinas para ordinais são mostradas abaixo.

Função	Descrição
Dec	Decrementa variável ordinal
Inc	Incrementa variável ordinal
Odd	Testa se um ordinal é ímpar
Pred	Predecessor do ordinal
Succ	Sucessor do ordinal
Ord	Ordem de um valor na faixa de valores de um tipo ordinal
Low	Valor mais baixo na faixa de valores
High	Valor mais alto na faixa de valores

Por exemplo, use o código abaixo no evento OnKeyPress de um Edit e veja o resultado.

```
Inc(Key);
```

Boolean

Variáveis do tipo Boolean podem receber os valores lógicos True ou False, verdadeiro ou falso. Uma variável Boolean ocupa 1 byte de memória.

TDateTime
O tipo TDateTime guarda data e hora em uma estrutura interna igual ao tipo Double, onde a parte inteira é o número de dias desde 31/12/1899 e a parte decimal guarda a hora, minuto, segundo e milissegundo. As datas podem ser somadas ou subtraídas normalmente.
Existem várias rotinas de manipulação de datas e horas, usadas com o tipo TDateTime, veja algumas abaixo.

Rotina	**Descrição**
Date	Retorna a data do sistema
Now	Retorna a data e hora do sistema
Time	Retorna a hora do sistema
DayOfWeek	Retorna o dia da semana de uma data especificada
DecodeDate	Decodifica um valor TDateTime em Words de dia, mês e ano
DecodeTime	Decodifica um valor TDateTime em Words de hora, minuto, segundo e milissegundos
EncodeDate	Retorna um TDateTime a partir de Words de dia, mês e ano
EncodeTime	Retorna um TDateTime a partir de Words de hora, minuto, segundo e milissegundos

No help de cada uma das funções acima você vai encontrar alguns exemplos, veja os colocados abaixo.

```
if DayOfWeek(Date) = 1 then
  ShowMessage('Hoje é Domingo, pé de cachimbo!')
else
  ShowMessage('Hoje não é Domingo, pé de cachimbo!');
var
  A, M, D: Word;
begin
  DecodeDate(Date, A, M, D);
```



```
  ShowMessage(Format('Dia %.2d do mês %.2d de %d.', [D, M,
A]));
end;
```

Variant
Tipo genérico, que pode atribuir e receber valores de qualquer outro tipo. Evite usar variáveis do tipo Variant, pois o uso dessas variáveis podem prejudicar a performance do programa, além de diminuir a legibilidade do código fonte e a integridade do executável, veja o trecho de código abaixo e note como esse tipo de variável tem um comportamento estranho.

```
var
  V1, V2, V3: Variant;
begin
  V1 := True;
  V2 := 1234.5678;
  V3 := Date;
  ShowMessage(V1 + V2 + V3);
end;
```

Conversões de Tipo
Freqüentemente você vai precisar converter um tipo de dado em outro, como um número em uma string. Para essas conversões você pode usar duas técnicas, o TypeCasting e as rotinas de conversão de tipos.

TypeCasting
TypeCast é uma conversão direta de tipo, usando o identificador do tipo destino como se fosse uma função. Como o Delphi não faz nenhuma verificação se a conversão é válida, você deve tomar um certo cuidado ao usar um TypeCast para não criar programas instáveis.

```
var
  I: Integer;
```

```
  C: Char;
  B: Boolean;
begin
  I := Integer('A');
  C := Char(48);
  B := Boolean(0);
  Application.MessageBox(PChar('Linguagem de Programação' + #13 + 'Delphi 3'), 'SENAC',
    MB_ICONEXCLAMATION);
end;
```

Rotinas de Conversão

As principais rotinas de conversão estão listadas na tabela abaixo. Caso você tente usar uma dessas rotinas em uma conversão inválida, pode ser gerada uma exceção.

Rotina	**Descrição**
Chr	Byte em Char
StrToInt	Integer em String
IntToStr	String em Integer, com um valor default caso haja erro
StrToIntDef	String em Integer, com um valor default caso haja erro
IntToHex	Número em String Hexadecimal
Round	Arredonda um número real em um Integer
Trunc	Trunca um número real em um Integer
StrToFloat	String em Real
FloatToStr	Real em string
FormatFloat	Número real em string usando uma string de formato
DateToStr	TDateTime em string de data, de acordo com as opções do Painel de Controle
StrToDate	String de data em TDateTime
TimeToStr	TDateTime em Strind de Hora
StrToTime	String de hora em TDateTime
DateTimeToStr	TDateTime em string de data e hora
StrToDateTime	String de data e hora em TDateTime
FormatDateTime	TDateTime em string usando uma string de formato
VarCast	Qualquer tipo em outro usando argumentos do tipo Variant
VarAsType	Variante em qualquer tipo
Val	String em número, real ou inteiro
Str	Número, real ou inteiro, em String

Veja alguns exemplos de como usar essas rotinas. Conversão de dados é uma operação muito comum na programação em Object Pascal, seeria interessante dar uma olhada no help de cada uma das funções acima.

```
var
  I: Integer;
  D: Double;
  S1, S2: string;
begin
  D := 10.5;
  I := Trunc(D);
  S1 := FloatToStr(D);
  S2 := IntToStr(I);
  ShowMessage(S1 + #13 + S2);
end;
```

```
var
  A, B, Soma: Bouble;
begin
  A := StrToFloat(EdtA.Text);
  B := StrToFloat(EdtB.Text);
  Soma := A + B;
  ShowMessage(Format('%f + %f = %f', [A, B, Soma]);
end;
```

Expressões
Uma expressão é qualquer combinação de operadores, variáveis, constantes, valores literais e chamadas de funções que resultem em um valor de determinado tipo. Uma expressão é usada sempre que precisamos de uma valor que possa ser obtido por uma expressão.

```
A + 12 * C
Date - 4
StrToInt(Edit1.Text + Edit2.Text)
StrToDate(Edit2.Text) - StrToDate(Edit1.Text)
12 * A / 100
A < B
```

Operadores
Os operadores são usados em expressões e a ordem em que as expressões são executadas depende da precedência desses operadores. Veja abaixo a lista de operadores em ordem descendente de precedência.

Operadores	**Descrição**
Operadores Unários	
@	Endereço
not	*Não* booleano ou bit voltado para *não*
Operadores Multiplicativos	
*	Multiplicação ou interseção de conjuntos
/	Divisão de Real
div	Divisão de Inteiro
mod	Resto de divisão de Inteiros
as	TypeCast seguro quanto ao tipo (RTTI)
and	*E* booleano ou bit voltado para *e*
shl	Deslocamento de bits à esquerda
shr	Deslocamento de bits à direita
Operadores Aditivos	
+	Adição ou união de conjuntos
-	Subtração ou diferença de conjuntos
or	*Ou* booleano ou bit voltado para *ou*
xor	*Ou* exclusivo booleano ou bit voltado para *ou* exclusivo

Para forçar uma expressão de menor precedência a ser executada antes, você pode usar os parênteses, como mostrado abaixo.

```
(5 - 2) * 3;
(A > B) and (A < C)
```

Para fazer potenciação, use a função *Power*, abaixo temos que A é igual a A elevado a 4.

```
A := Power(A, 4);
```

Estruturas de Decisão
If
O if é uma estrutura de decisão usada para realizar instruções em determinadas condições. O if é considerado uma só instrução, por isso, só encontramos o ponto-e-vírgula no final. O else é opcional.

```
if Opn.Execute then
  Img.Picture.LoadFromFile(Opn.FileName);

if Nota < 5 then
  ShowMessage('Reprovado')
else
  ShowMessage('Aprovado');
```

Case

Permite que o fluxo da execução seja desviado em função de várias condições de acordo com o valor do argumento, que tem que ser ordinal, caso o valor do argumento não corresponda a nenhum dos valores listados, podemos incluir um else.

```
case Ch of
  ' ': ShowMessage('Espaço');
  '0'..'9': ShowMessage('Dígito');
  '+', '-', '*', '/': ShowMessage('Operador');
else
  ShowMessage('Caractere especial');
end;
case CbbBorda.ItemIndex of
  0: BorderStyle := bsDialog;
  1: BorderStyle := bsSingle;
  2: BorderStyle := bsSizeable;
end;
```

Estruturas de Repetição

While

O laço while executa uma instrução até que uma condição seja falsa.

```
I := 10;
while I >= 0 do
begin
  ShowMessage(IntToStr(I));
  Dec(I);
end;
```

For

O laço for executa uma instrução um número determinado de vezes, incrementando uma variável de controle automaticamente a cada iteração. Caso seja preciso que a contagem seja decremental, pode-se usar *downto* em vez de *to*.

```
for I := 1 to ComponentCount do
  ShowMessage('O ' + IntToStr(I) + 'º Componente é ' + Components[I - 1].Name);
for I := Length(Edit1.Text) downto 1 do
  ShowMessage(Edit1.Text[I]);
```

Repeat

O laço repeat executa instruções até que uma condição seja verdadeira.

```
I := 1;
repeat
  S := InputBox('Acesso', 'Digite a senha', '');
  Inc(I);
  if I > 3 then
    Halt;
until S = 'fluminense';
```

Quebras de Laço

Em qualquer um dos laços mostrados podemos usar o procedimento Break para cancelar a repetição e sair do laço, podemos também forçar a próxima iteração com o procedimento Continue.

```
I := 1;
while true do
begin
  Inc(I);
  if I < 10000000 then
    Continue;
  ShowMessage('Chegamos a dez milhões');
  Break;
end;
```

Tipos Definidos Pelo Usuário

O usuário também pode declarar tipos não definidos pelo Delphi. Essas declarações são feitas na seção *type*, da interface ou implementation, sendo que na implementation esses tipos não poderão ser usados em outras Units.
Dificilmente você terá que definir tipos, a não ser classes, pois os tipos padrão do Delphi são o bastante para a maioria das aplicações.

Strings Limitadas
Caso se deseje limitar o número de caracteres que uma string pode receber, podemos criar um tipo de string limitada.

```
TNome = string[40];
TEstado = string[2];
```

Tipo Sub-Faixa
É um subconjunto de um tipo ordinal e possui as mesmas propriedades do tipo original.

```
TMaiusculas = 'A'..'Z';
TMes = 1..12;
```

Enumerações
Define uma seqüência de identificadores como valores válidos para o tipo. A cada elemento da lista de identificadores é associado internamente um número inteiro, iniciando pelo número 0, por isso são chamados de tipos enumerados.

```
TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
TDiaSemana = (Seg, Ter, Qua, Qui, Sex, Sab, Dom);
```

Ponteiros
Ponteiros armazenam endereços de memória, todas as classes em Object Pascal são implementadas como ponteiros, mas raramente o programador vai precisar usá-los como tal.

```
TIntPtr: ^Integer;
```

Records
O tipo record é uma forma de criar uma única estrutura com valores de diferentes tipos de dados. Cada um dos dados de um record é chamado de campo.

```
TData = record
  Ano: Integer;
  Mes: TMes;
  Dia: Byte;
end;
var
  Festa: TData;
begin
  Festa.Ano := 1997;
  Festa.Mes := Mai;
  Festa.Dia := 8;
end;
```

Arrays
Arrays fornecem uma forma de criar variáveis que contenham múltiplos valores, como em uma lista ou tabela, cujos elementos são do mesmo tipo. Veja abaixo alguns exemplos de arrays de dimensões variadas.

```
TTempDia = array [1..24] of Integer;
TTempMes = array [1..31, 1..24] of Integer;
TTempAno = array [1..12, 1..31, 1..24] of Integer;
var
  TD: TTempDia;
  I: Integer;
begin
  for I := 1 to 24 do
    TD[I] := StrToIntDef(InputBox('Temperaturas', 'Digite a temperatura na hora '
      + IntToStr(I), ''), 30);
end;
```

Um array pode ser definido como constante tipada, onde todos os seus elementos devem ser inicializados.

```
FAT: array[1..7] of Integer = (1, 2, 6, 24, 120, 720, 5040);
```

O tipo dos elementos de um array pode ser qualquer um, você pode ter uma array de objetos, de conjuntos, de qualquer tipo que quiser, até mesmo um array de arrays.

```
TTempMes = array [1..31] of TTempDia;
TBtnList = array [1..10] of TButton;
```

Sets

São conjuntos de dados de um mesmo tipo, sem ordem, como os conjuntos matemáticos. Conjuntos podem conter apenas valores ordinais, o menor que um elemento pode assumir é zero e o maior, 255.

```
TBorderIcons = set of TBorderIcon;
BorderIcons := [biSystemMenu, biMinimize];
if MesAtual in [Jul, Jan, Fev] then
  ShowMessage('Férias');
```

Os conjuntos podem ser definidos como constantes ou constantes tipadas, como abaixo.

```
DIG_HEXA = ['0'..'9', 'A'..'Z', 'a'..'z'];
DIG_HEXA: set of Char = ['0'..'9', 'A'..'Z', 'a'..'z'];
```

Procedimentos, Funções e Métodos

As ações de um objeto devem ser definidas como métodos. Quando a ação não pertence a um objeto, como uma transformação de tipo, essa ação deve ser implementada em forma de procedimentos e/ou funções.

Procedimentos

Procedimentos são sub-rotinas, que realizam uma tarefa e não retornam um valor. A declaração de um procedimento é feita na seção interface e a definição, na seção implementation. Ao chamar o identificador do procedimento, com os parâmetros necessários, esse procedimento será executado. Veja abaixo o exemplo de uma unit com a implementação um procedimento.

```
unit Tools;
interface
procedure ErrorMsg(const Msg: string);
implementation
uses Forms, Windows;
 Procedure ErrorMsg(const Msg: string);
begin
  Application.MessageBox(PChar(Msg), 'Operação inválida', MB_ICONERROR);
end;
 end.
```

Funções

Funções são muito semelhantes a procedimentos a única diferença é que as funções retornam um valor. O tipo do valor de retorno deve ser informado no cabeçalho da função. Na implementação da função deve-se atribuir o valor de retorno à palavra reservada *Result* ou ao identificador da função. Pode-se então usar a função em expressões, atribuições, como parâmetros para outras funções, em qualquer lugar onde o seu valor possa ser usado.

```
function Average(A, B: Double): Double;
begin
  Result := (A + B) / 2;
end;
```

Métodos

Métodos são funções ou procedimentos que pertencem a alguma classe, passando a fazer parte de qualquer objeto dessa classe. Na implementação de um método precisamos indicar qual a classe à qual ele pertence. Para chamar um método em algum lugar não pertencente à sua classe, como procedimentos, funções ou métodos de outras classes, deve ser indicado o objeto que deve executar o método. Os métodos usam os mesmos níveis de encapsulamento dos atributos.

```
type
  TFrmMsg = class(TForm)
    LblMsg: TLabel;
    BtnOk: TButton;
    BtnCancelar: TButton;
    ImgMsg: TImage;
  public
    procedure ShowMsg(const Msg: string);
  end;

procedure TFormMsg.ShowMsg(const Msg: string);
begin
  LblMsg.Caption := Msg;
  ShowModal;
end;
```

Parâmetros

Existem três tipos de passagem de parâmetros, que devem ser indicados na declaração da função ou procedimento. Parâmetros de tipos diferentes de vem ser separados por ponto e vírgula.

```
function MultiStr(const S: string;  N: Double; var Erro: Integer): string;
```

- Quando não é indicado o tipo de passagem, é passado o valor do parâmetro, como constante.
- Ao usar a palavra-chave *var*, não será enviado o valor do parâmetro e sim uma referência a ele, tornando possível mudar o valor do parâmetro no código do procedimento.
- Como alternativa você pode passar um parâmetro por referência constante, para isso use a palavra const antes da declaração do parâmetro.

With

Usado para facilitar o acesso às propriedades e métodos de um objeto.

```
with Edt do
begin
  CharCase := ecUpperCase;
  MaxLenght := 10;
  PasswordChar := '*';
  Text := 'Brasil';
end;
```

Self

Self é usado quando se quer referenciar a instância atual da classe. Se você precisar referenciar a instância atual de uma classe, é preferível usar Self em vez de usar o identificador de um Objeto, isso faz com que o código continue funcionando para as demais instâncias da classe e em seus descendentes.

Criando e Destruindo Objetos

Antes de tudo, você deve declarar o objeto, se quiser referenciá-lo. Para criá-lo, use o método Create, que é um método de classe. Para você usar um método de classe, referencie a classe, não o Objeto, como mostrado abaixo.

```
var
  Btn: TBitBtn;
begin
  Btn := TBitBtn.Create(Self);
  With Btn do
  begin
    Parent := Self;
    Kind := bkClose;
    Caption := '&Sair';
    Left := Self.ClientWidth - Width - 8;
    Top := Self.ClientHeight - Height - 8;
  end;
```

```
end;
```
Porém, se você não precisar referenciar o Objeto, poderia criar uma instância sem referência.
```
with TBitBtn.Create(Self) do
begin
  Parent := Self;
  Kind := bkClose;
  Caption := '&Sair';
  Left := Self.ClientWidth - Width - 8;
  Top := Self.ClientHeight - Height - 8;
end;
```
Para destruir um objeto, use o método Free. Para Forms, é recomendado usar o Release, para que todos os eventos sejam chamados.
O parâmetro do método Create é usado apenas em Componentes, para identificar o componente dono. Ao criar Forms, poderíamos usar o Objeto Application.
```
FrmSobre := TFrmSobre.Create(Application);
FrmSobre.ShowModal;
FrmSobre.Release;
```
Para criar objetos não componentes, você não precisa de nenhum parâmetro no método Create.
```
var
  Lst: TStringList;
begin
  Lst := TStringList.Create;
  Lst.Add('Alô, Teresinha!');
  Lst.Add('Uhh uhh...');
  Lst.SaveToFile('Teresinha.txt');
  Lst.Free;
end;
```