
Normas para codificação

Delphi

Versão 2.03



SUMÁRIO

| | |
|---|----|
| Sumário | 1 |
| Introdução | 2 |
| 1. IDE E SINTAXE | 3 |
| 1.1. Indentação | 3 |
| 1.2. Margens | 3 |
| 1.3. Par <i>begin..end</i> | 4 |
| 1.4. Parênteses | 5 |
| 1.5. <i>Char Case</i> para palavras reservadas e tipos primitivos | 5 |
| 2. COMANDOS | 7 |
| 2.1. Comando <i>if</i> | 7 |
| 2.2. Comando <i>case</i> | 7 |
| 2.3. Comando <i>for</i> | 8 |
| 2.4. Comando <i>while</i> | 9 |
| 2.5. Comando <i>repeat</i> | 10 |
| 2.6. Comando <i>Exit</i> | 10 |
| 2.7. Comando <i>with</i> | 10 |
| 3. EXCEÇÕES | 11 |
| 3.1. Tratamento de Exceções | 11 |
| 4. <i>Camel Case</i> | 13 |
| 4.1. Definição | 13 |
| 5. CONSTANTES E VARIÁVEIS GLOBAIS | 14 |
| 5.1. Constantes | 14 |
| 5.2. Variáveis Globais | 14 |
| 6. TIPOS SIMPLES | 15 |
| 6.1. Nomenclatura para derivações | 15 |
| 6.2. Tipos enumerados | 15 |
| 6.3. Tipos de ponto flutuante | 15 |
| 7. CLASSES | 16 |
| 7.1. Nomenclatura | 16 |
| 7.2. Escopos de visibilidade | 16 |
| 7.3. Atributos ou <i>fields</i> | 16 |
| 7.4. Métodos | 17 |
| 7.5. Propriedades | 18 |
| 8. COMPONENTES | 20 |
| 8.1. Padronização por tipo de componente | 20 |
| 8.2. Nomenclatura | 20 |
| 9. ARQUIVOS | 21 |
| 9.1. <i>Units</i> de código | 21 |
| 9.2. <i>Units</i> de interface ou formulários | 21 |
| 9.3. <i>Units</i> de <i>Data Module</i> | 21 |

INTRODUÇÃO

Este documento tem como objetivo estabelecer regras para a escrita de código Delphi, definindo padrões quanto a diversos aspectos intrínsecos à tarefa de desenvolvimento. Ao seguir este documento, a equipe de desenvolvimento estará apta a escrever códigos mais claros, fazendo com que seus membros possam facilmente entender códigos escritos por outros integrantes da equipe.

1. IDE E SINTAXE

1.1. Indentação

- 1.1.1. A indentação é de dois espaços por nível. Não guarde caracteres *tab* nos seus arquivos fonte.

1.2. Margens

- 1.2.1. A margem direita deve ser ajustada para 120 caracteres;
- 1.2.2. Comandos que se estenderem além da margem direita devem ser quebrados em quantas linhas forem necessárias para sua correta acomodação, dentro das margens;
- 1.2.3. As linhas que se originarem da quebra de um comando devem sofrer indentação de dois espaços em relação à primeira linha do comando.

Exemplo:

| | |
|--|--|
| <pre>// Incorreto: Ultrapassou a margem direita Usuario.InserirEndereco('Rua Santos, 52', 'Ipatininguinha' 'Centro'); // Correto! Usuario.InserirEndereco('Rua Santos, 52', 'Ipatininguinha', 'Centro'); // Correto! Usuario.InserirEndereco('Rua Santos, 52', 'Ipatin' + 'inguinha', 'Centro');</pre> | |
|--|--|

- 1.2.4. Cláusulas que inicializam seções devem ser separadas da seção anterior por uma linha em branco.

Exemplo:

```
unit uUsuario;
// Linha em branco separando a seção
uses
    TObjectBase;
// Linha em branco separando a seção
const
    TAMANHO_MAXIMO_ENDERECO = 128;
// Linha em branco separando a seção
type
    TUsuario = class(TObjectBase);
    public
        procedure InserirEndereco(const pEndereco, pCidade, pBairro: string);
    end;
// Linha em branco separando a seção
implementation
// Linha em branco separando a seção
uses
    System.SysUtils;
```

1.3. Par *begin..end*

- 1.3.1. Todo novo bloco de código, iniciado por um comando *if*, *while*, *for*, etc. deve estar contido em um par *begin..end*, mesmo que este contenha apenas uma linha;
- 1.3.2. A palavra reservada *begin* deve aparecer em sua própria linha, sem indentação;
 - 1.3.2.1. Exceções permitidas:
 - A palavra *begin* não deve estar em uma nova linha quando precedida da palavra reservada *else*. Neste caso, o comando *else* também deve estar na mesma linha da palavra *end* que finaliza o bloco anterior (*then*);
 - A palavra *begin* deve ser indentada quando iniciar um bloco de uma cláusula *case*.

Exemplo:

```
// Incorreto: o begin deve estar em uma nova linha
for lLoop := 1 to 10 do begin
    ...
end;
```

```
// Correto: o begin está em sua própria linha
for lLoop := 1 to 10 do
begin
    ...
end;
```

```
// Incorreto: o comando if deve iniciar um novo bloco com o par begin..end
if lAtivo then
    ...
```

```
// Incorreto: o begin deve estar em uma nova linha
if lAtivo then begin
    ...
end;
```

```
// Correto: o comando if iniciou um novo bloco com o par begin..end
if lAtivo then
begin
    ...
end;
```

```
// Incorreto: comandos end else begin devem estar na mesma linha
if lAtivo then
begin
...
end
else
begin
...
end;

// Correto: os comandos end else begin estão na mesma linha
if lAtivo then
begin
...
end else begin
...
end;
```

1.4. Parênteses

- 1.4.1. Não deve haver espaços em branco entre um abre parênteses e o próximo caractere, salvo quando for necessário quebrar uma linha, de forma que o comando seja corretamente acomodado entre a margem esquerda e direita;
- 1.4.2. Não deve haver espaços em branco entre um fecha parênteses e o caractere anterior;
- 1.4.3. Não deve haver espaços entre um abre parênteses e o nome da *procedure* ou *function* que ocasionaram a abertura de parênteses, salvo quando for necessário quebrar uma linha, de forma que o comando seja acomodado entre a margem esquerda e direita.

Exemplo:

```
// Incorreto: não deve haver espaços entre os parênteses e os operandos
Result := ( Base * Altura ) / 2;

// Correto: sem espaços entre os parênteses e os operandos
Result := (Base * Altura) / 2;
```

```
// Incorreto: não deve haver espaços entre os parênteses e o parâmetro
ShowMessage ( 'Teste' );

// Incorreto: não deve haver espaços entre a procedure e a
// passagem de parâmetros
ShowMessage ('Teste');
```

```
// Correto!
ShowMessage ('Teste');
```

1.5. Char Case para palavras reservadas e tipos primitivos

- 1.5.1. Palavras reservadas, como *string*, *function*, *array*, etc. (grifadas em negrito), devem ser escritas com todos os caracteres minúsculos;

- 1.5.1.1. A exceção a esta regra está no procedimento *Register*;
- 1.5.2. Tipos primitivos, como *Integer*, *Char*, *Double*, etc., devem respeitar suas declarações originais.

Exemplo:

```
class function TCalculadora.Somar(const pElementos: array of Integer): Integer;
var
    lElemento: Integer;
begin
    Result := 0;

    for lElemento in pElementos do
    begin
        Inc(Result, lElemento);
    end;
end;
```

2. COMANDOS

2.1. Comando *if*

- 2.1.1. Se a expressão booleana a ser testada for composta por duas ou mais condições, estas devem estar dispostas da esquerda para a direita, em ordem inversa à sua complexidade computacional. Esta regra resultará em uma otimização de código, uma vez que o compilador finalizará o teste quando uma das condições for suficiente para determinar o resultado de toda a expressão.

Exemplo:

```
if FATivo and (GetIdade < IDADE_MAXIMA_CLIENTE) then
begin
    // Neste caso, a resolução da primeira condição é computacionalmente
    // mais simples que a resolução da segunda condição. Se FATivo for
    // falso, todo o processamento da segunda condição é desnecessário e
    // não será executado. Desta forma, o programa ganha em performance.
end;
```

- 2.1.2. Sempre que o encadeamento sucessivo de comandos *if* puder ser substituído por um comando *case*, então um *case* deve ser utilizado.

2.2. Comando *case*

- 2.2.1. Os valores contidos pelo comando *case* devem estar ordenados de forma crescente;
- 2.2.2. Cada caso deve ser indentado em relação ao comando *case*;
- 2.2.3. A implementação para cada caso deve começar em uma nova linha indentada;
- 2.2.4. A implementação decorrente de um caso poderá omitir o par *begin..end* quando esta contemplar apenas um comando. Caso contrário, o par *begin..end* é exigido pelo compilador, e neste caso, o par *begin..end* deverá sofrer indentação, e seu código interno será novamente indentado;
- 2.2.5. Blocos de comandos decorrentes da implementação de um caso não devem ter mais que cinco linhas, incluindo as linhas do par *begin..end*;
- 2.2.6. Caso a cláusula *else* seja necessária, esta deve estar alinhada ao comando *case*, não sofrendo indentação. O código contido pela cláusula *else* deverá ser indentado normalmente.

Exemplo:

```

case Valor of
  1..10: ProcedimentoA; // Incorreto: implementação na mesma linha do caso
  11..20:
    ProcedimentoB;      // Incorreto: comando não está indentado
  21,
  23,                    // Incorreto: a lista de valores deve estar na
  25:                    // mesma linha
    ProcedimentoC;
  30:
    begin                // Incorreto: par begin..end não está indentado
      ProcedimentoD;
      ProcedimentoE;
    end;
  31:
    begin
      ProcedimentoF;      // Incorreto: comandos contidos pelo par
      ProcedimentoG;      // begin..end não estão indentados
    end;
  32:
    begin                // 1
      ProcedimentoH;      // 2
      ProcedimentoI;      // 3
      ProcedimentoJ;      // 4
      ProcedimentoK;      // 5 - Incorreto: contagem do bloco
    end;                // 6 - superou 5 linhas
    else                  // Incorreto: cláusula else está indentada
      raise Exception.Create('Valor inesperado no case de exemplo.');
```

```

end;

case Valor of
  1..10:
    ProcedimentoA;      // Correto: comando decorrente está em uma nova linha
  11..20:
    ProcedimentoB;      // Correto: comando está indentado
  21, 23, 25:           // Correto: os valores estão em uma mesma linha
    ProcedimentoC;
  30:
    begin                // Correto: par begin..end está indentado
      ProcedimentoD;
      ProcedimentoE;
    end;
  31:
    begin
      ProcedimentoF;      // Correto: comandos contidos pelo par begin..end
      ProcedimentoG;      // estão indentados
    end;
  32:                    // Correto: criado novo método para simplificar
    ProcedimentoHIJK;    // a leitura do bloco case
  else                  // Correto: cláusula else não está indentada
    raise Exception.Create('Valor inesperado no case de exemplo.');
```

```

end;
```

2.3. Comando for

2.3.1. Comandos *for* devem ser utilizados em *loops* com número definido de iterações;

Exemplo:

```
for lI := 0 to FClientes.Count - 1 do
begin
  lCliente := GetCliente(lI);

  if lCliente.EnderecoAtivo then
  begin
    lCliente.GerarCorrespondencia;
  end;
end;
```

2.3.2. Quebras extraordinárias de *loops* pelo comando *Break* dificultam a compreensão do código fonte, e consequentemente seu uso configura uma má prática de programação. Toda quebra de *loop* deve estar prevista na condição de saída do mesmo. O Uso do comando *Break* está proibido!

Exemplo:

```
for lI := 0 to FClientes.Count - 1 do
begin
  lCliente := GetCliente(lI);

  if lCliente.Nome = pNomePesquisa then
  begin
    Result := lCliente;
    Break; // Incorreto!
  end;
end;
```

2.3.3. Assim como o comando *Break*, o comando *Continue* gera um desvio de código que dificulta a sua compreensão. O uso do comando *Continue* está proibido!

2.4. Comando *while*

2.4.1. Da mesma forma que o comando *if*, caso um comando *while* possua duas ou mais condições de quebra, estas devem estar dispostas da esquerda para a direita, em ordem inversa a sua complexidade computacional;

2.4.2. Todas as condições de quebra do *loop* devem estar contidas na cláusula *while*;

2.4.3. O uso dos comandos *Break* e *Continue* está proibido!

Exemplo:

```
lI := 0;
Result := nil;
while not Assigned(Result) and (lI < FClientes.Count) do // Correto!
begin
    lCliente := GetCliente(lI);

    if lCliente.Nome = pNomePesquisa then
    begin
        Result := lCliente;
    end;
end;
```

2.5. Comando *repeat*

- 2.5.1. Da mesma forma que o comando *if* e *while*, o comando *repeat* deve ter suas condições de quebra ordenadas da condição de menor complexidade para a condição de maior complexidade;
- 2.5.2. *Loops* do tipo *repeat* devem ser utilizados sempre que o *loop* exija no mínimo uma iteração, mas com número final de iterações indefinido;
- 2.5.3. Todas as condições de quebra do *loop* devem estar contidas na cláusula *until*;
- 2.5.4. O uso dos comandos *Break* e *Continue* está proibido!

Exemplo:

```
lTentativas := 0;
Result := False;
repeat
    Result := FRecursosRede.CopiarArquivo(pOrigem, pDestino);

    if not Result then
    begin
        Inc(lTentativas);
    end;
until Result or (lTentativas = MAXIMO_TENTATIVAS);
```

2.6. Comando *Exit*

- 2.6.1. O uso do comando *Exit* configura uma má prática de programação e está proibido!

2.7. Comando *with*

- 2.7.1. O uso do comando *with* configura uma má prática de programação e está proibido!

3. EXCEÇÕES

3.1. Tratamento de Exceções

- 3.1.1. O tratamento de exceções deve ser fortemente utilizado, tanto para correção de erros, quanto para proteção de recursos.
- 3.1.2. Recursos cuja liberação seja de responsabilidade do desenvolvedor, devem ter a garantia de liberação assegurada em código, através da utilização do comando *try..finally*;
- 3.1.3. Não se deve utilizar um comando *try..finally* para desalocar mais de um recurso ao mesmo tempo, por não se tratar de uma prática segura;

Exemplo:

```
lFornecedor := TFornecedor.Create(pCodigoFornecedor);
lCliente := TCliente.Create(pCodigoCliente); // Incorreto!
try
    ...
finally
    lCliente.Free;
    lFornecedor.Free; //Incorreto!
end;

// Correto!
lFornecedor := TFornecedor.Create(pCodigoFornecedor);
try
    lCliente := TCliente.Create(pCodigoCliente);
    try
        ...
    finally
        lCliente.Free;
    end;
finally
    lFornecedor.Free;
end;
```

- 3.1.4. Não se deve utilizar o comando *try..except* para simplesmente exibir mensagens de erro, esta tarefa é de responsabilidade do tratador padrão de exceções da aplicação. O comando *try..except* deve ser utilizado quando realmente necessitamos de uma reação como consequência a um erro, esperado ou inesperado;

Exemplo:

```
// Incorreto!
try
    lQuery.ExecSQL;
except
    ShowMessage('Aconteceu um erro ao atualizar um registro!');
    Abort;
end;

// Correto!
try
    lQuery.ExecSQL;
except
    on E: Exception do
    begin
        E.RaiseOuterException(Exception.Create('Erro ao atualizar o registro!'));
    end;
end;

// Correto!
lTransacao.Begin;
try
    lQuery1.ExecSQL;
    lQuery2.ExecSQL;
    lTransacao.Commit;
except
    lTransacao.Rollback;
    raise;
end;
```

4. Camel Case

4.1. Definição

- 4.1.1. *Camel Case* é o nome de um método para escrita de identificadores compostos por mais de uma palavra. Para facilitar a visualização, o método propõe que as palavras devem ser iniciadas com letras maiúsculas, e as demais permanecem minúsculas;
- 4.1.1.1. Quando um identificador for nomeado com uma sigla, a sigla deve permanecer com todas as suas letras maiúsculas;
- 4.1.2. Não é permitido o uso do caractere *underline* ('_');

Exemplo:

```
// Incorreto: Todas as palavras devem começar por letras maiúsculas
Nomecliente := 'João da Silva';
// Incorreto: Todas as palavras devem começar por letras maiúsculas
nomecliente := 'João da Silva';
// Incorreto: Uso do underline
Nome_Cliente := 'João da Silva';
// Incorreto: Somente a primeira letra de cada palavra deve ser maiúscula
NOMECLIENTE := 'João da Silva';
// Correto!
NomeCliente := 'João da Silva';

// Incorreto: Sigla composta por letras minúsculas
CpfCliente := '123.456.789-00';
// Correto!
CPFCliente := '123.456.789-00';
```

5. CONSTANTES E VARIÁVEIS GLOBAIS

5.1. Constantes

- 5.1.1. O uso de constantes globais é desaconselhado. Sempre que possível, constantes devem ser declaradas dentro da classe ou método apropriado;
- 5.1.2. Constantes devem ser batizadas com nomes que expressem facilmente o seu propósito e devem ser grafadas em CAIXA ALTA. Quando o nome de uma constante for composto por mais de uma palavra, estas devem estar separadas por *underscores*;

Exemplo:

```
// Declaração de constante em local desaconselhado
NUMERO_MAXIMO_TENTATIVAS = 3;

TLogin = class
strict private const
    // Declaração de constante em local apropriado
    NUMERO_MAXIMO_TENTATIVAS = 3;
    ...
```

5.2. Variáveis Globais

- 5.2.1. Variáveis globais estão proibidas! *Class vars* devem ser utilizados como alternativas às variáveis globais;

6. TIPOS SIMPLES

6.1. Nomenclatura para derivações

- 6.1.1. Tipos derivados simples devem ser prefixados com a letra 'T' maiúscula;
 - 6.1.1.1. A exceção a esta regra está na nomenclatura de ponteiros, que devem ser prefixados com a letra 'P' maiúscula;
- 6.1.2. Após ser prefixado, o nome dado ao tipo deve seguir o método *Camel Case*;

Exemplo:

```
TListaInteiros = array of Integer;  
PResultadoPesquisa = ^TResultadoPesquisa;  
TNotaAluno = Double;  
TDiaDaSemana = (dsSegunda, dsTerca, dsQuarta, dsQuinta, dsSexta);
```

6.2. Tipos enumerados

- 6.2.1. Os itens de um tipo enumerado devem ser prefixados com duas ou mais letras, minúsculas, que sirvam como mnemônico ao tipo enumerado;
- 6.2.2. Depois de prefixado, o item deve ser batizado seguindo o método *Camel Case*;
- 6.2.3. A enumeração dos itens deve acontecer de forma que o delimitador (',') fique junto ao item anterior, seguido de um espaço que precederá o próximo item;

Exemplo:

```
// Incorreto: itens do enumerado não estão prefixados com o mnemônico  
TDiaDaSemana = (Segunda, Terca, Quarta, Quinta, Sexta);  
  
// Incorreto: delimitador não está junto ao item anterior  
TDiaDaSemana = (dsSegunda , dsTerca , dsQuarta , dsQuinta , dsSexta);  
  
// Incorreto: não foi observado o espaço que precede os itens  
TDiaDaSemana = (dsSegunda,dsTerca,dsQuarta,dsQuinta,dsSexta);  
  
// Correto!  
TDiaDaSemana = (dsSegunda, dsTerca, dsQuarta, dsQuinta, dsSexta);
```

6.3. Tipos de ponto flutuante

- 6.3.1. O tipo *Real* não deve ser utilizado, pois o mesmo foi substituído pelo tipo *Double* e existe somente para manter a compatibilidade com o Pascal;
- 6.3.2. O uso do tipo *Extended* é permitido somente em casos estritamente necessários. Seu uso é desencorajado por possuir um tamanho de dado não otimizado para os barramentos dos processadores;
- 6.3.3. Sempre que possível, o tipo *Double* deve ser substituído pelo tipo *Currency*, de forma a evitar problemas de arredondamento durante operações aritméticas;

7. CLASSES

7.1. Nomenclatura

- 7.1.1. Nomes de classes devem ser prefixados com a letra 'T' maiúscula;
 - 7.1.1.1. A exceção a esta regra está na herança (direta ou indireta) da classe *Exception*. Neste caso, o nome da nova exceção deve ser prefixado com a letra 'E' maiúscula;
- 7.1.2. Depois de prefixadas, as classes devem ser batizadas seguindo o método *Camel Case*;

7.2. Escopos de visibilidade

- 7.2.1. Os escopos de visibilidade declarados em uma classe devem estar, sempre que possível, ordenados do escopo mais restritivo ao menos restritivo;

Exemplo:

```
TProduto = class
strict private
    // Escopo estritamente privado
private
    // Escopo privado
strict protected
    // Escopo estritamente protegido
protected
    // Escopo protegido
public
    // Escopo público
published
    // Escopo publicável
end;
```

7.3. Atributos ou *fields*

- 7.3.1. Os *fields* de uma classe devem obrigatoriamente pertencer ao escopo *private* ou *strict private*, sendo *strict private* o escopo preferencial para declaração de *fields*;
- 7.3.2. Todos os *fields* devem ser prefixados com a letra 'F' maiúscula;
- 7.3.3. Depois de prefixados, os *fields* devem ser batizados seguindo o método *Camel Case*;

Exemplo:

```

TProduto = class
strict private
  ValorProduto: Currency; // Incorreto: Field não prefixado
  FValorProduto: Currency; // Correto!
private
  Valor_Produto: Currency; // Incorreto: Uso do caractere underline
  FValorProduto: Currency; // Correto!
strict protected
  FValorProduto: Currency; // Incorreto: Field declarado em escopo indevido
protected
  FValorProduto: Currency; // Incorreto: Field declarado em escopo indevido
public
  FValorProduto: Currency; // Incorreto: Field declarado em escopo indevido
published
  FValorProduto: Currency; // Incorreto: Field declarado em escopo indevido
end;

```

7.4. Métodos

- 7.4.1. Métodos devem receber nomes significativos, de forma a facilitar a interpretação de suas funcionalidades;
- 7.4.2. O verbo que representa a ação do método deve estar sempre no infinitivo;
- 7.4.3. Métodos devem ser batizados seguindo o método *Camel Case*;
- 7.4.4. Ao criar uma *function*, o tipo de retorno deve ser declarado de forma que o caractere ':' esteja junto ao *token* anterior, e sucedido por um espaço (que precederá o próximo *token*);

Exemplo:

```

TPessoaFisica = class
public
  // Desaconselhável: nome do método não deixa a sua funcionalidade explícita
  // (o que será validado?)
  function Validar: Boolean;
  // Incorreto: Verbo não está no infinitivo
  function ValidaCamposObrigatorios: Boolean;
  // Incorreto: Caractere ':' está precedido por um espaço
  function ValidarCamposObrigatorios : Boolean;
  // Incorreto: Caractere ':' não está sucedido por um espaço
  function ValidarCamposObrigatorios:Boolean;

  // Correto!
  function ValidarCamposObrigatorios: Boolean;
end;

```

- 7.4.5. Rotinas utilizadas para a leitura e escrita de propriedades (*Getters* e *Setters*) deverão ser prefixadas com *Get* e *Set*, respectivamente.

Exemplo:

```

property Nome: string read FNome write SetNome;
property Idade: Byte read GetIdade;

```

- 7.4.6. Parâmetros devem ser prefixados com a letra 'p' minúscula;
- 7.4.7. Parâmetros não devem apresentar prefixos de tipagem ('s' para *string*, 'i' para *Integer*, etc.);
- 7.4.8. Depois de prefixados, parâmetros devem ser batizados seguindo o método *Camel Case*;
- 7.4.9. Os delimitadores utilizados na declaração de parâmetros (',', ';;', ':') devem estar juntos ao *token* que os precede, e sucedidos por um espaço (que precederá o próximo *token*);
- 7.4.10. Sempre que a regra do método permitir, parâmetros devem receber a diretiva *const*;

Exemplo:

```
// Incorreto: parâmetros não estão prefixados com a letra 'p'
function Somar(const Operando1, Operando2: Integer): Integer;
// Incorreto: uso de tipagem nos parâmetros
function Somar(const piOperando1, piOperando2: Integer): Integer;
// Incorreto: Delimitadores estão precedidos por espaço
function Somar(const pOperando1 , pOperando2 : Integer): Integer;
// Incorreto: Delimitadores não estão sucedidos por espaço
function Somar(const pOperando1,pOperando2:Integer):Integer;
// Correto!
function Somar(const pOperando1, pOperando2: Integer): Integer;
```

- 7.4.11. Sempre que a regra de uma *function* permitir, seu resultado deve ser escrito diretamente dentro da variável *Result*, evitando o uso de variáveis auxiliares;

Exemplo:

```
function TPessoa.GetValorTotalPedidos: Currency;
var
  lI: Integer
begin
  Result := 0;

  for lI := 0 to FPedidos.Count - 1 do
  begin
    Result := Result + FPedidos[lI].Valor;
  end;
end;
```

- 7.4.12. Variáveis locais devem ser prefixadas com a letra 'l' ('L') minúscula;
- 7.4.13. Variáveis não devem apresentar prefixos de tipagem ('s' para *string*, 'i' para *Integer*, etc.);
- 7.4.14. Depois de prefixadas, as variáveis devem ser batizadas seguindo o método *Camel Case*;
- 7.4.15. Os delimitadores utilizados na declaração de variáveis (',', ':') devem estar juntos ao *token* que os precede, e sucedidos por um espaço (que precederá o próximo *token*);

7.5. Propriedades

- 7.5.1. Propriedades devem ser batizadas seguindo o método *Camel Case*;

7.5.2. Propriedades podem ser criadas a partir do escopo *strict protected*;

8. COMPONENTES

8.1. Padronização por tipo de componente

- 8.1.1. Todo componente utilizado no desenvolvimento de aplicações deverá receber um mnemônico único, composto por três letras minúsculas;

8.2. Nomenclatura

- 8.2.1. Todo componente inserido no formulário em tempo de *design* deve ser renomeado. Nenhum componente deve ficar com o nome dado pelo IDE;
- 8.2.2. Todo componente inserido no formulário deverá ser prefixado com o seu respectivo mnemônico;
- 8.2.3. Depois de prefixado, o componente deve ser batizado seguindo o método *Camel Case*;

9. ARQUIVOS

9.1. *Units* de código

- 9.1.1. *Units* de código devem ser prefixadas com a letra 'u' minúscula;
- 9.1.2. Depois de prefixada, a *unit* deve ser batizada seguindo o método *Camel Case*, com um nome que expresse a principal funcionalidade da mesma.
 - 9.1.2.1. No caso de uma *unit* cuja principal funcionalidade seja a implementação de uma classe, recomenda-se que esta *unit* receba o nome da classe sem seu prefixo 'T';

9.2. *Units* de interface ou formulários

- 9.2.1. *Units* de interface, utilizadas para gerenciar um formulário, devem ser prefixadas com a letra 'f' minúscula;
- 9.2.2. Depois de prefixada, a *unit* deve ser batizada seguindo o método *Camel Case*, com o nome do formulário sem seu prefixo (identificador da classe);

9.3. *Units* de *Data Module*

- 9.3.1. *Units* utilizadas para gerenciar um *Data Module*, devem ser prefixadas com a letra 'd' minúscula;
- 9.3.2. Depois de prefixada, a *unit* deve ser batizada seguindo o método *Camel Case*, com o nome do formulário sem seu prefixo (identificador da classe);

Aquasoft Tecnologia da Informação
www.aquasoft.com.br