

# RECURSOS AVANÇADOS NO DELPHI



Thulio  
Bittencourt

**RECURSOS  
AVANÇADOS  
NO DELPHI**



**{CLASS  
OPERATOR}**



# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

Fala ai Radizeiros e Radizeiras, tudo bem com vocês?

No Delphi, é possível alterar a maneira de como um operador trabalha (para tipos definidos pelo desenvolvedor).

Esse recurso no Delphi permite ao programador redefinir o significado de um operador, que é conhecimento como **Class Operator**.

*Class Operator* existe na linguagem Delphi a muito tempo, só que eles eram específicos para record.

Com a evolução do Delphi esses *Class Operator* foram expandido para classes propriamente ditas.

Isso resultou no fato do Delphi compilar para arquitetura ARM, e dessa forma eles foram expandidos para classes, mas para o Windows, ele não funciona para classe.

Mas ele funciona com *records*, e você consegue fazer chover com *Class Operator*.

Então vamos lá ao nosso primeiro exemplo, do poder do *Class Operator*.

Possuo uma classe record, que possui um valor inteiro e um string.

Então vamos trabalhar com essa nossa classe record.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
1 TProduto = record
2   Valor : Integer;
3   Nome : String;
4 end;
5 ...
6 procedure TForm1.Button1Click(Sender: TObject);
7 var
8   a, b, c : TProduto;
9 begin
10   a.Valor := 10;
11   b.Valor := 30;
12   c := a + b;
13   ShowMessage(IntToStr(c.Valor));
14 end;
```

Observe que eu tenho um Nome que é uma string e um Valor que é um inteiro.

Criei três variáveis do tipo *record TProduto*, onde *a.valor* recebe 10, e o produto *b* recebe 30.

Logo abaixo estou informando que *c* irá receber a soma desses dois produtos, conseguiu entender e pegar qual é a questão?

Bom, a situação é a seguinte, primeiro essa implementação o compilador não teria que compilar, se eu criar duas classes e tentar soma-las não irá funcionar, correto?

As classes não se somam, os *records* não se somam, mas nesse nosso exemplo estou solicitando para eles se somarem, e olha que interessante.

Eu mandei somando duas instâncias de uma classe, mas o correto seria eu somar os valores e não a classe.

```
1 ...
2   c.valor := a.valor + b.valor;
3 ...
```

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
while not dmDados.SQLQuery1.Eof do
begin
  Operador := TOperador();
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```

Fazendo dessa forma teríamos a soma correta.

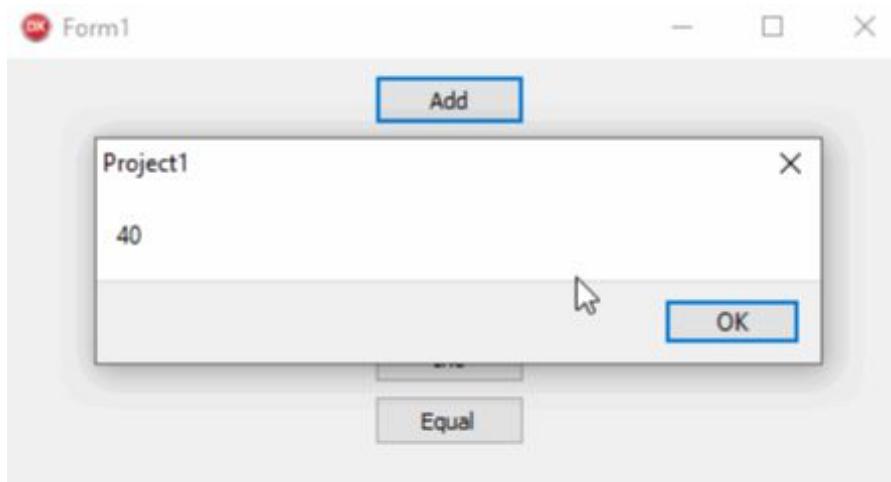
Mas no nosso exemplo estamos somando dois objetos que são uma instância da classe *TProduto*.

Não estou somando as propriedades da classe, e sim os objetos da instância da classe.

Logo em seguida eu estou mandando visualizar o resultado no valor de “c”.

Olha o que irá acontecer, primeiro que não é para compilar.

Mas observe que ele irá compilar, e não só irá compilar, que ao clicar no **Add** ele irá mostrar a soma de “a” e “b”.



# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

Usando *Class Operator*, implicitamente, conseguimos somar objetos.

Imagine se você tivesse o item da nota fiscal?

Na hora de fazer o total da nota, e você pegar todos os itens e mandar somar, você não precisa varrer todos os itens para ver o que tem que ser somado.

Você só vai somar o objeto, e já irá ter uma classe implícita que faz isso para você de forma automática.

Você observou que eu não estou chamando nenhuma classe, não chamo um método, e nada, simplesmente mandei dois objetos.

**Porque que isso é possível?**

Algo que devemos ter como primordial para o conhecimento é a documentação da linguagem.

E esse é um recurso que está disponível lá na documentação do Delphi que são os **Class Operator**

[http://docwiki.embarcadero.com/RADStudio/Rio/en/Operator\\_Overloading\\_\(Delphi\).](http://docwiki.embarcadero.com/RADStudio/Rio/en/Operator_Overloading_(Delphi).)

Existem vários operadores que permitem ser sobre carregados do Delphi.

O que aconteceu nesse nosso exemplo, é que não é sobre carregado um método, mas sim o operador.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

No exemplo, nós sobrecrevemos os operadores do Delphi, para fazer uma função do Delphi, diferente daquilo que queríamos que ela fizesse.

Com isso passa ser possível fazer operações que antes não era possível.

Vamos ver como isso funciona.

Então temos uma classe record, e dentro dessa classe record temos os Class Operator, o operador de adição, o **Add**, que está dentro da documentação do Delphi.

Esse é um dos diversos operadores que podemos utilizar.

Então como funciona isso dentro do Delphi.

```
1 TProduto
2   Valor : Integer;
3   Nome : String;
4   class operator Add(a, b: TProduto) : TProduto;
5 ...
```

Nessa implementação eu estou dizendo que ele recebe os dois tipos e retorna o novo tipo.

Então eu estou dizendo, que todas as vezes que alguém usar o operador **Add**, ou seja, o operador de soma, para somar dois objetos do tipo *TProduto*, esse método do código acima, será chamado.

E o que esse método está fazendo?

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
1 class operator TProduto.Add(a, b: TProduto): TProduto;
2 begin
3     Result.Valor := a.Valor + b.Valor;
4 end;
```

Ela pega o resultado que irá ter da soma dos valores de “a” e “b” implicitamente.

Então para quem está desenvolvendo não precisa fazer esse cálculo, ele simplesmente irá fazer a soma de a e b, e aí pronto, implicitamente ele pega os valores e faz a forma, que seria “correta”, que a maioria das pessoas fazem hoje.

Muito legal não é mesmo?

Dessa forma você pode fazer o seu Delphi somar a classe, ou subtrair a classe.

Por exemplo, você tem uma rotina de venda, que ela tem a lista de itens, faz todo o cálculo da venda.

Vamos dizer assim, *TVenda + TItem*, e implicitamente você irá pegar essa classe de itens que chegou preenchida e vai fazer o que tiver que fazer para adicionar uma venda, recalcular total e por aí vai.

Quando alguém cancelar uma venda, por exemplo, você irá fazer *TVenda - (item removido)*, a classe vem implicitamente e faz todo o cálculo para você.

Então você consegue deixar o código mais transparente, mais legível, para quem for trabalhar não precisa ficar usando um monte de typecasting na tela, não precisa fazer um monte de coisas, você encapsula as suas funcionalidades.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
while not dmDados.QueryUserLogado.EOF do begin
  Operador := TOperador();
  Operador.Nome := dmDados.QueryUserLogado.FieldByName('NOME').AsString;
```

A segunda, e uma outra operação que temos dentro das *Class Operator*, é o **Implicit**.

```
1 procedure TForm1.Button3Click(Sender: TObject);
2 var
3   a : TProduto;
4 begin
5   a := '30';
6   ShowMessage(IntToStr(a.Valor));
7 end;
```

O que acontece dentro desse botão?

Observe, eu tenho uma variável do tipo *TProduto*, que foi chamada de “a”, eu digo que “a” irá receber uma string de 30.

Prestou atenção neste código?

A variável está recebendo uma string, eu to dizendo que uma variável do tipo de uma classe record está recebendo uma string, só que eu não estou fazendo assim:

```
1 ...
2 a.Valor := StrToInt('30');
3 ...
```

# RECURSOS AVANÇADOS NO DELPHI

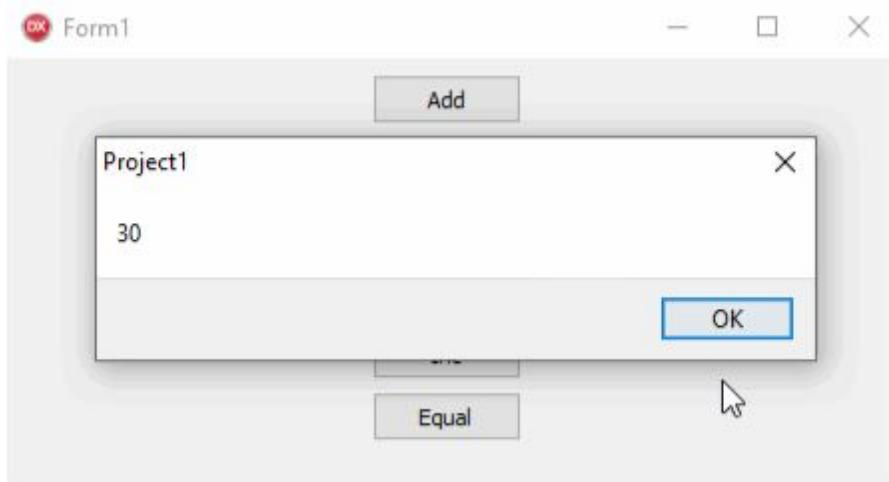
## {CLASS OPERATOR}

```
while not dmDados.QueryUsuarios.Eof do  
begin  
  Operador := TOperador();  
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```

Esse seria o correto, o natural da programação para fazer com que o valor string 30 seja adicionado a uma property inteira dentro da classe record.

Gracias ao *Class Operator*, o record recebe uma string de 30.

E ele vai retornar o valor, vamos ver isso funcionando?



Viu que ele retornou tudo certinho e sem erro?

Observe que ele implicitamente está pegando esse valor, fazendo a conversão desse valor e jogando para dentro da propriedade que está recebendo o valor.

Mas você deve estar se perguntando como isso está sendo feito.

Foi criado um outro *Class Operator* dentro dessa nossa classe record, que é o **Implicit**.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
1 TProduto
2     Valor : Integer;
3     Nome : String;
4     class operator Add(a, b: TProduto) : TProduto;
5     class operator Implicit(a : String) : TProduto;
6 ...
7     class operator TProduto.Implicit(a : String) : TProduto;
8 begin
9     Result.Valor := StrToInt(a);
10 end;
```

O que seria esse operador implícito, e o que ele faz?

Ele recebe um valor, de um tipo de objeto diferente, eu te dizendo que esse *record* agora é um record mutante.

Então eu tenho uma variável do tipo *TProduto*, teoricamente, ela só poderia receber objeto do mesmo tipo dela, mas graças ao **Implicit** eu posso fazer com que o *TProduto* receba qualquer tipo de variável, contando que ela tenha o *Class Operator Implicit* tratando esse tipo.

Se eu tentar ao invés de passar uma string e passar para ele um inteiro, o compilador irá gerar erro, pois estou tratando somente string.

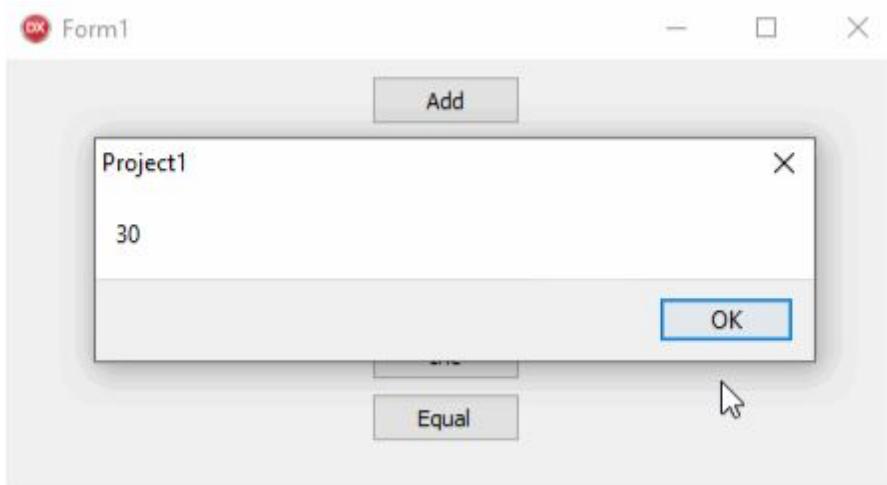
Mas caso eu queira também tratar isso, basta criar um outro *Class Operator Implicit*, recebendo por parâmetro um tipo inteiro.

```
1 ...
2     class operator Implicit(a : Integer) : TProduto;
3 ...
4     class operator TProduto.Implicit(a : Integer) : TProduto;
5 begin
6     Result.Valor := a;
7 end;
8 ...
9 procedure TForm8.Button3Click(Sender: TObject);
10 var
11     a : TProduto;
12 begin
13     a := 30;
14     ShowMessage(IntToStr(a.Valor));
15 end;
```

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

Observe que agora estou passando para a variável “a” um inteiro.



Viu que não tem erro nenhum na compilação, e claro na execução do exemplo.

Isso acontece pelo fato de estar informando para o compilador que tenho um método que trata valores inteiros também.

Quando chegar no objeto *TProduto* um valor inteiro, ele irá chamar implicitamente o método criado, e fazer o que tem que ser feito.

Essa é uma outra funcionalidade do *Class Operator*.

Isso é muito legal, porque, se passar para ele um inteiro ou uma string o compilador irá compreender e tratar isso, sem gerar erro.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

Muito legal né?

Temos também um outro recurso sobre *Class Operator*, que se chama **positive**, que em algumas linguagens, na hora de fazer um incremento, simplesmente adicionamos o sinal de “+” na frente de uma variável.

E com *Class Operator* você pode sobrepor os operadores de incremento e decremento.

```
1 procedure TForm1.Button3Click(Sender: TObject);
2 var
3     Prod : TProduto;
4 begin
5     Prod := 10;
6     Prod := +Prod;
7     ShowMessage(IntToStr(Prod.Valor));
8 end;
```

Observe que possuo uma variável do tipo *TProduto*, dizendo que o *prod* está recebendo 10, e logo em seguida passando para o *prod* um incremento, ou seja, *+prod*.

Geralmente qual seria o correto para esta ação?

```
1 ...
2 Prod.Valor := Prod.Valor + 1;
3 ...
```

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

Esse seria o normal, e claro, é o que muito de nós que programamos em Delphi estamos acostumados.

Só que agora estou desmistificando isso da seguinte forma, estou dizendo que o objeto de *TProduto*, não a propriedade valor de *TProduto*, e sim um objeto, está recebendo um objeto com o operador de adição, ou seja, fazendo o incremento no objeto.

Nesse ponto interceptamos o operador, também com *Class Operator*, para tratarmos o que queremos que aconteça quando essa ação acontece.

Vamos dizer que você tem sua classe de itens, e o número dos itens são incrementados, todas as vezes você vai adicionar um item vai colocando, item1, item2...

Você não precisa ficar colocando o número do item +1, só fazer dessa forma que acabei de mostrar que já incrementa o número.

E como fazemos isso no *Class Operator*?

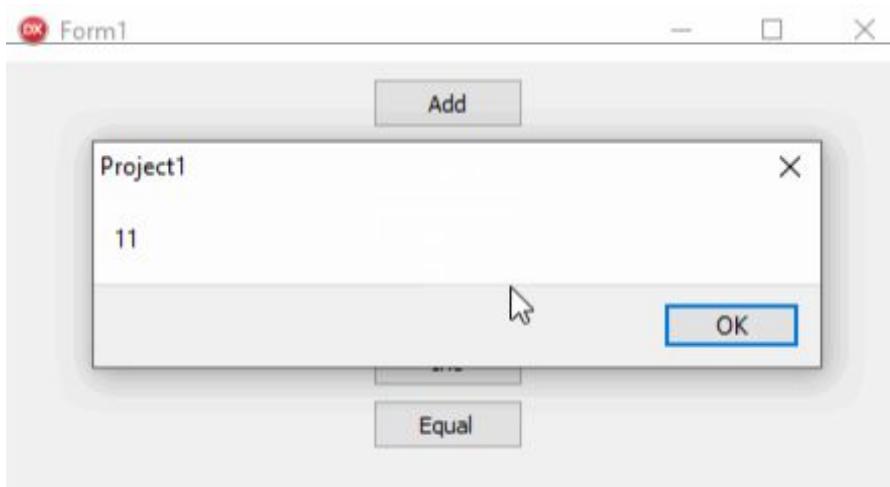
```
1 ...  
2 class operator Positive(Value : TProduto) : TProduto;  
3 ...  
4 class operator TProduto.Positive(Value : TProduto) : TProduto;  
5 begin  
6     Result.Valor := Value.Valor + 1;  
7 end;
```

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
while not dmDados.SQLQuery1.Eof do
begin
  Operador := TOperador();
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```

Observe que dentro desse método **positive**, quando o objeto **positive** recebe um **TProduto**, ou seja, um positivo de **TProduto**, o retorno irá setar o parametro **value.valor +1**, incrementando o valor passado e irá resultar no incremento do mesmo.



Viu como é bem simples esse incremento?

Isso tudo acontece implicitamente, onde o *Class Operator* sobrecarrega para nós.

A mesma coisa acontece com o decremento.

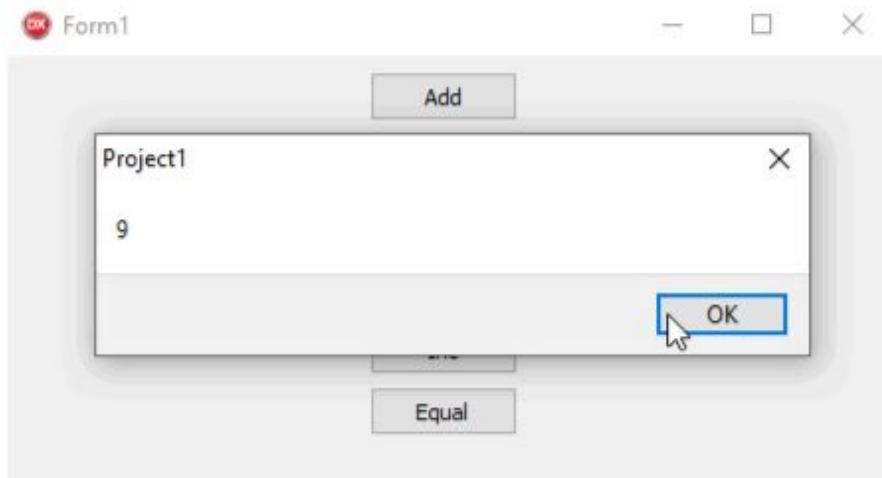
```
1 ...  
2 class operator Negative(Value : TProduto) : TProduto;  
3 ...  
4 class operator TProduto.Negative(Value : TProduto) : TProduto;  
5 begin  
6   Result.Valor := Value.Valor - 1;  
7 end;  
8 ...  
9 procedure TForm1.Button4Click(Sender : TObject);  
10 var  
11   Prod : TProduto;  
12 begin  
13   Prod := 10;  
14   Prod := -Prod;  
15  
16   ShowMessage(IntToStr(Prod.Valor));  
17 end;
```

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
while not dmDados.SQLQuery1.Eof do
begin
  Operador := TOperador();
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```

Assim como foi feito no incremento, fazemos com o decremento.



Muito legal não é mesmo?

Outro recurso que podemos sobrestrar é o **Inc**.

```
1 procedure TForm1.Button5Click(Sender: TObject);
2 var
3   Prod : TProduto;
4 begin
5   Prod := 10;
6   Inc(Prod);
7   ShowMessage(IntToStr(Prod.Valor));
8 end;
```

Você deve estar bem familiarizado com o **Inc**, não é verdade?

É muito utilizado esse operador, e com o *Class Operator* podemos também sobrestrar-lo.

E se eu quiser dar um **Inc** somente na minha classe?

E lá dentro tratar qual o valor que irá ser incrementado?

Basta ir em nossa classe e implementar o *Class Operator* de **Inc**.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
1 ...
2 class operator Inc(Value : TProduto) : TProduto;
3 ...
4 class operator TProduto.Inc(Value : TProduto) : TProduto;
5 begin
6     Result.Valor := Value.Valor + 10;
7 end;
```

Observe que criamos o operador de incremento, onde todas as vezes que alguém incrementar com o *TProduto*, irá incrementar mais 10.

Eu to dizendo que o meu **Inc** não é mais 1, e sim mais 10.

Mas observe que a passagem de parâmetro do método, não estou passando um valor inteiro, e sim a classe *TProduto*.

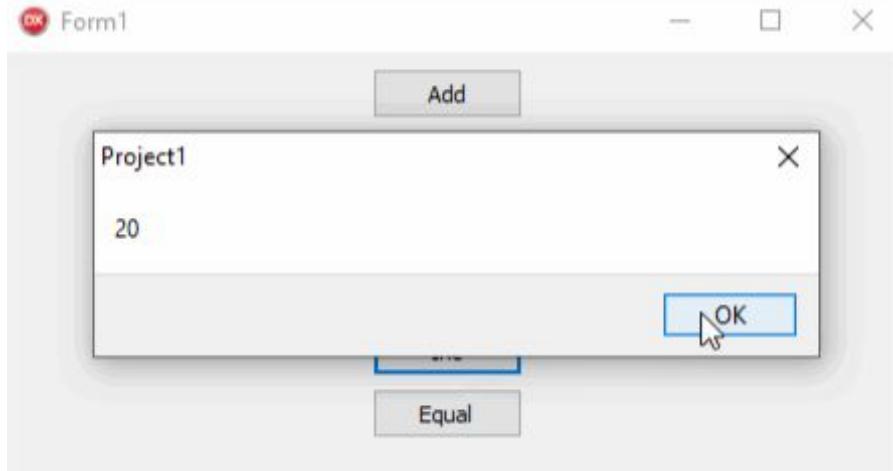
Observe que não estou passando **Inc** no objeto inteiro e sim na classe.

```
1 procedure TForm1.Button5Click(Sender: TObject);
2 var
3     Prod : TProduto;
4 begin
5     Prod := 10;
6     Inc(Prod);
7     ShowMessage(IntToStr(Prod.Valor));
8 end;
```

Estou passando um valor para a classe, e logo em seguida estou incrementando o objeto *prod*, implicitamente ele vai incrementar no atributo valor.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}



Como eu disse que era 10, executei o `Inc` que está sobrecarregado com mais 10, então ele me retornou 20.

**E se eu quiser um decremento de 2?**

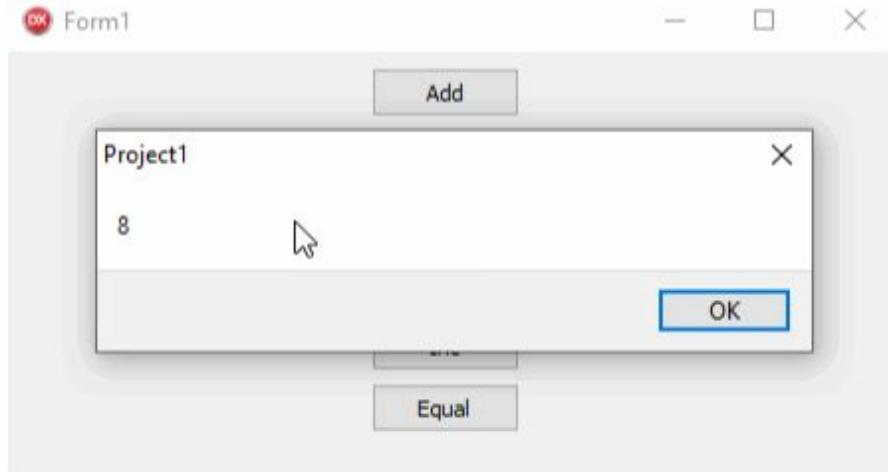
É só tratar dentro do método `Negative`, que ele irá decrementar o valor que você definir.

```
1 class operator TProduto.Negative(Value : TProduto) : TProduto;
2 begin
3     Result.Valor := Value.Valor - 2;
4 end;
```

Agora todas as vezes que alguém fizer um decremento o `negative` ele vai decrementar 2.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}



E agora o último que iremos ver aqui é o operador *Equal*.

Vamos verificar primeiro antes sem o *Equal* para ver o comportamento do compilador sem ele.

```
1 procedure TForm1.Button6Click(sender: TObject);
2 var
3     ProdA, ProdB : TProduto;
4 begin
5     ProdA.Nome := 'Arroz';
6     ProdA.Valor := 10;
7     ProdB.Nome := 'Feijão';
8     ProdB.Valor := 20;
9
10    if ProdA = ProdB then
11        ShowMessage('Sucesso')
12    else
13        ShowMessage('Erro');
14 end;
```

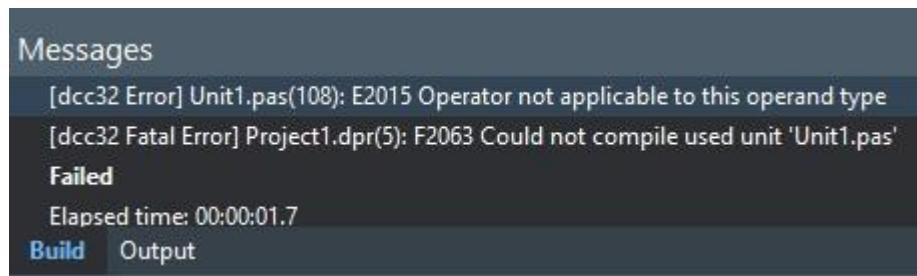
# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

Observe que criei dois produtos, *ProdA* e *ProdB*, no *ProdA* eu coloquei *Nome Arroz*, e *Valor 10*, e no *ProdB*, *Nome Feijão*, e *Valor* coloquei 20.

E estou fazendo uma estrutura condicional, onde verifico se *ProdA* é igual a *ProdB*.

Primeira coisa que acontece quando compilamos isso no Delphi, é retornado um erro.



Observe que o erro está informando que o operador não é aplicado ao tipo de objeto.

Ele está informando que não tem capacidade de comparar se o *ProdA* é igual ao *ProdB*, ele não é reconhecido, desta forma eu não tenho como comparar esses dois objetos, não consigo ver se esses dois objetos se um é igual ao outro.

Com a utilização do *Class Operator* é possível fazermos isso.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}

```
1 ...  
2 class operator Equal(A, B: TProduto) : Boolean;  
3 ...  
4 class operator TProduto.Equal(A, B: TProduto) : Boolean;  
5 begin  
6     Result := (A.Valor = B.Valor) and (A.Nome = B.Nome);  
7 end;
```

Nós possuímos o atributo *Equal* do *Class Operator*, que é o atributo de igual.

Todas as vezes que alguém chamar o operador igual, e se minha classe possuir um *Class Operator* sobrecregando esse operador, ele irá chamar esse método que criamos na classe de produto, onde é retornado verdadeiro ou falso o que foi tratado dentro desse método.

Se observar a documentação da Embarcadero você irá compreender como é trabalhado esse operador.

Do mesmo jeito você consegue fazer para diversos operadores dentro do Delphi.

Simplesmente dentro do método de **Equal** eu verifico se os valores completos são iguais, os *Nomes* e *Valores*, tanto para *A* quanto para *B*.

# RECURSOS AVANÇADOS NO DELPHI

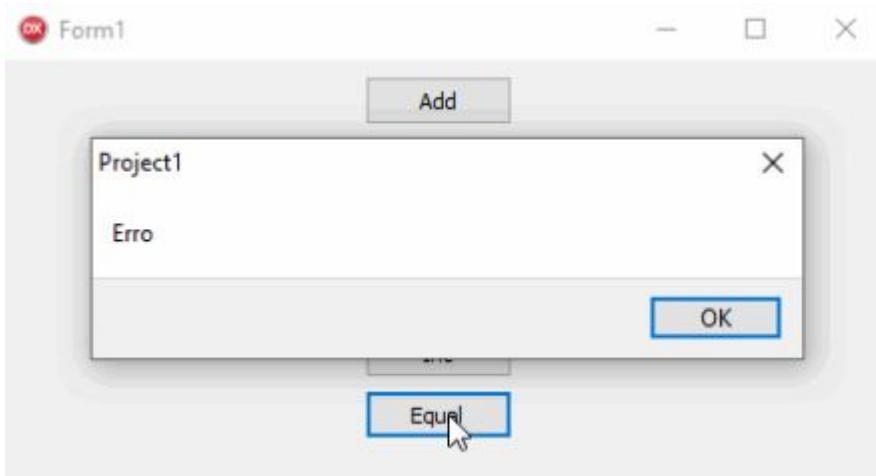
## {CLASS OPERATOR}

```
while not dmDados.QueryUsuarios.Eof do
begin
  Operador := TOperador();
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```

Agora a minha classe, tem a capacidade de dizer que dois atributos records do tipo *TProduto* são iguais ou não.

O que acabamos de fazer pode ser feito para qualquer classe sua, basta você criar o método de comparação.

No código que implementamos do botão de *Equal*, ele faz a verificação, vamos ver se irá mostrar a mensagem da verificação.

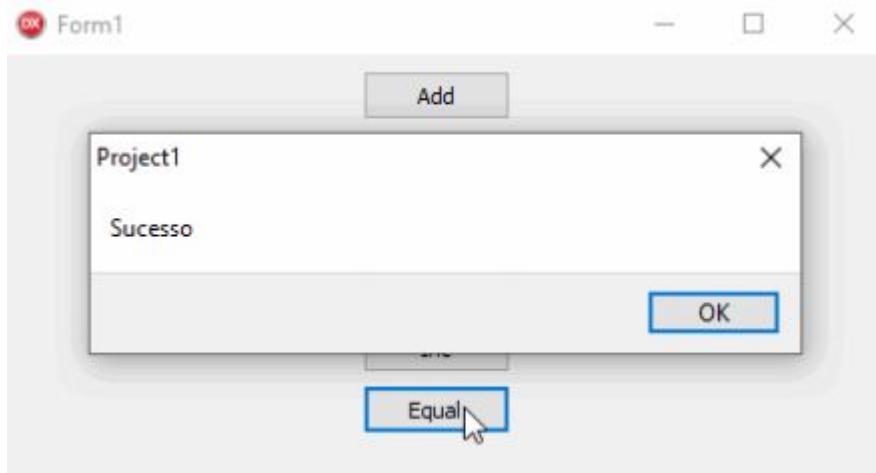


Observe que me foi retornado a mensagem de erro, isso ocorreu porque os valores de *Nome* são diferentes, onde *ProdA* = *Arroz*, e *ProdB* é *Feijão*.

Ele conseguiu comprar isso para nós e saber que ele são diferentes, agora vou igualar os valores e vamos ver o resultado.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS OPERATOR}



Agora ele comparou certinho.

Criamos então uma funcionalidade a mais que não existia.

Agora você conhece o que são os *Class Operator*, que é um recurso da linguagem, e você pode explorar a documentação do Delphi.

Existem vários operadores, e com todos esses operadores você pode trabalhar e sobrepor eles para suas classes não primitivas do Delphi.

Devemos nos atentar a um detalhe, o *Class Operator* não funciona na classe em si, quando for rodar para Windows, essa funcionalidade do *Class Operator* só foi adicionada para compiladores ARM, ou seja, para os compiladores Mobile, dessa forma você pode colocar os operadores nas classes.

# RECURSOS AVANÇADOS NO DELPHI



# {ABSOLUTE}

# RECURSOS AVANÇADOS NO DELPHI

## {ABSOLUTE}

```
while not dmDados.QueryUserLogado.EOF do
begin
  Operador := TOperador.Create;
  Operador.Nome := dmDados.QueryUserLogado.FieldByName('NOME').AsString;
```

Imagine uma situação na qual temos um **Edit** e um **Memo** dentro de um formulário e queremos que tudo o que for escrito neles apareça em caixa-alta. Imagine a princípio também que, para atender essa funcionalidade, injetamos no evento **onChange** de cada um desses objetos rotinas para realizar essa operação.

Até o momento parece simples e fácil né?  
Mas pare pra pensar o quanto se tornaria custoso em todo o seu projeto?

Um monte de casting dentro do seu projeto para realizar essas operações, e por que não deixar mais limpo seu código?

Irei falar sobre **absolute**, é um negócio muito legal que tem dentro do Delphi, mas que poucas pessoas usam.

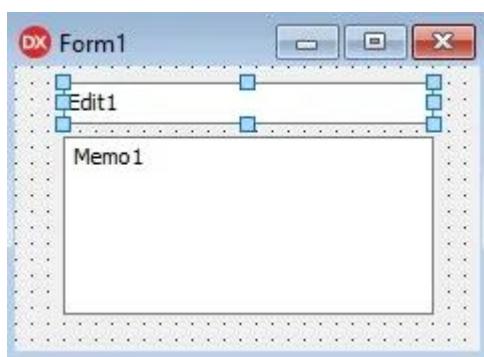
De acordo com os arquivos de ajuda do Delphi, o modificador **absolute** é uma palavra reservada que torna possível que uma nova variável resida no mesmo endereço de memória de outra variável facilitando type-casting.

# RECURSOS AVANÇADOS NO DELPHI

## {ABSOLUTE}

Realmente vale muito apena você conhecer esses diversos recursos que temos dentro do Delphi.

Então vamos lá em nosso exemplo bem simples.



Nesse exemplo, eu quero que tudo que eu digitar fique maiúsculo, é um exemplo muito simples não é?

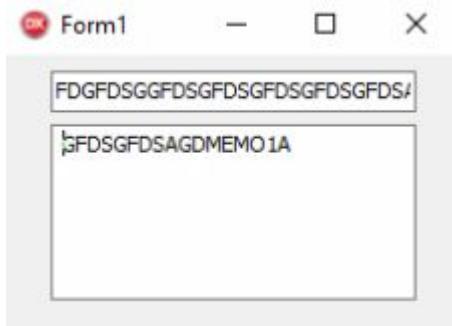
```
1 procedure TForm1.Edit1Change(Sender: TObject);
2 begin
3     UpperNormal(Sender);
4     //UpperAbsolute(Sender);
5 end;
6 procedure TForm1.Memo1Change(Sender: TObject);
7 begin
8     UpperNormal(Sender);
9     //UpperAbsolute(Sender);
10 end;
```

Como é um código simples vamos executar o exemplo e ver se ao digitar está colocando as informações em caixa alta.

# RECURSOS AVANÇADOS NO DELPHI

## {ABSOLUTE}

```
while not dmDados.QueryUsuarios.EOF do begin
  Operador := TOperador.Create;
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```



Viu como que tudo que digitamos ficou maiúsculo?  
Até aí nenhuma diferença não é verdade?

Vamos dentro do código desse método *UpperNormal*.

```
1 procedure TForm1.UpperNormal(Sender: TObject);
2 begin
3   if Sender is TEdit then
4     TEdit(sender).Text := Uppercase(TEdit(Sender).Text);
5   if Sender is TMemo then
6     TMemo(sender).Text := Uppercase(TEdit(Sender).Text);
7 end;
```

Dentro do evento do *onChange* desses componentes, a cada alteração é chamado o método *UpperNormal*, passando *Sender*, ou seja, o próprio objeto em si.

Ao receber o objeto, e o método trata esse objeto, se *Sender* for um *Edit*, é feito um *casting* nele, acessando a propriedade *Text*, que recebe um *Uppercase* no *casting* de *Text*, da mesma forma com o *Memo*.

# RECURSOS AVANÇADOS NO DELPHI

## {ABSOLUTE}

Fazendo assim transformamos os textos em maiúsculo.

Mas até aqui nada de novo, não é verdade?

É aí que entra o **Absolute**, ele veio para eliminar esses *casting*, com o **Absolute** nós fazemos é um *casting* na declaração da variável.

Vou trocar agora o código dos botões, onde iremos executar o *UpperAnsolute*.

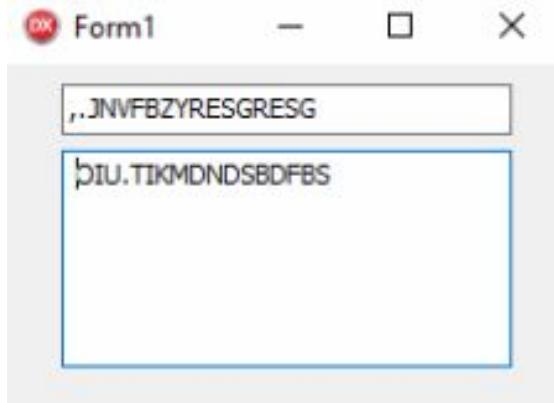
```
1 procedure TForm1.Edit1Change(Sender: TObject);
2 begin
3     //UpperNormal(Sender);
4     UpperAbsolute(Sender);
5 end;
6 procedure TForm1.Memo1Change(Sender: TObject);
7 begin
8     //UpperNormal(Sender);
9     UpperAbsolute(Sender);
10 end;
```

Viu como falei anteriormente?

O **Absolute** é muito simples e rápido, e seu conceito é muito legal e poucas pessoas o utilizar.

# RECURSOS AVANÇADOS NO DELPHI

## {ABSOLUTE}



Viu que ele fez praticamente a mesma coisa que a execução anterior.

Vamos ver agora como funciona esse *UpperAbsolute*?

```
1 procedure TForm1.UpperAbsolute(Sender: TObject);
2 var
3     Edit : TEdit absolute Sender;
4     Memo : TMemo absolute Sender;
5 begin
6     if Sender is TEdit then
7         Edit.Text := Uppercase(Edit.Text)
8     else
9         Memo.Text := Uppercase(Memo.Text);
10 end;
```

Você já pode reparar que não posso na implementação os *casting*, simplesmente é verificado se o *Sender* é um *TEdit*, passando o valor maiúsculo para a variável *Edit*, assim também para o *Memo*.

Observou que não tem um *casting* dentro da implementação como fizemos anteriormente?

Isso acontece porque o **Absolute**, faz esse “*casting*” direto na declaração da variável.

# RECURSOS AVANÇADOS NO DELPHI

## {ABSOLUTE}

Na tipificação da variável é passado o **absolute**, para que o compilador entenda que ao receber o *Sender* desses objetos seja feito o “casting” implicitamente.

Fazendo assim, estou apontando para o endereço da memória onde está o *Sender*, assumindo, lógico, o comportamento do objeto tipificado, por exemplo, *TEdit*.

A partir desse momento tenho uma variável *Edit*, que é do tipo *TEdit*, que está apontando para o endereço de memória onde está o *Sender*.

Dentro da implementação do método podemos trabalhar sem necessidade de *casting*.

Não preciso mais fazer *casting* na implementação do código, somente declaro na variável e pronto, e só trabalhar com ele.

Muito simples, e rápido de usar, e deixa seu código mais limpo.

# RECURSOS AVANÇADOS NO DELPHI

## {ABSOLUTE}

```
while not dmDados.QUserados.Eof do
begin
  Operador := TOperador.Create;
  Operador.Nome := dmDados.QUserados.FieldByName('NOME').AsString;
```

Só uma atenção, e ter um pouco de cuidado pois você estará trabalhando com ponteiro, implicitamente ele está apontando para algum lugar na memória, e se não tiver cuidado poderá ter problemas.

Mas o Delphi nessa parte de trabalhar com ponteiros é muito tranquilo, muito mais light.

Quando conhecemos os recursos da linguagem conseguimos eliminar muito lixo dentro do nosso código, fazendo com que ele se torne mais legível e de melhor compreensão para manutenções futuras.

A seguir:

## {CONHECENDO AS RECORDS VARIANTS}

**RECURSOS  
AVANÇADOS  
NO DELPHI**



**{CONHECENDO  
AS RECORDS  
VARIANTS}**

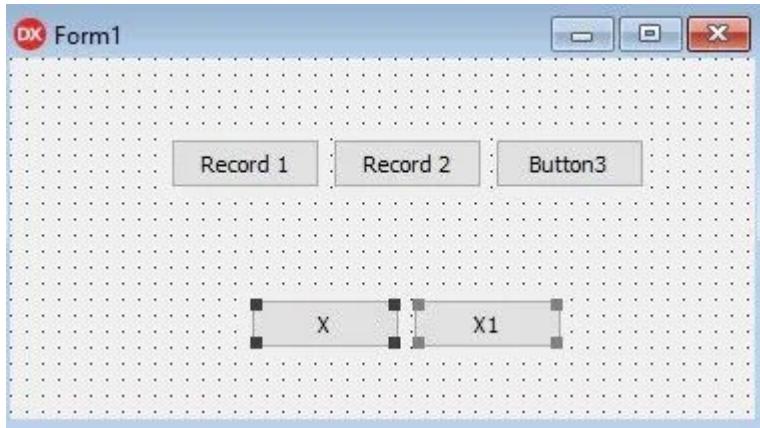


# RECURSOS AVANÇADOS NO DELPHI

## {CONHECENDO AS RECORDS VARIANTS}

Você já ouviu falar, ou já utilizou Records Variants?

Se não, vem que irei lhe mostrar como podemos trabalhar com elas, e claro, a praticidade e simplicidade da utilização.  
Eu tenho aqui o meu exemplo.



Temos um exemplo aqui bem simples, e o que são essas *Records Variants* e qual é o objetivo delas?

Antes de explicar a fundo sobre elas, vale a pena saber que são recursos disponíveis na linguagem.

Antigamente nós tínhamos dois problemas muito grande que era armazenamento e processamento.

E antigamente tínhamos pouquíssimos recursos de memórias, e gerenciar memória da forma mais produtiva possível era um diferencial para a linguagem.

# RECURSOS AVANÇADOS NO DELPHI

## {CONHECENDO AS RECORDS VARIANTS}

As Records Variantes estão aí para isso, só para contextualizar vocês nisso.

Então vamos ver o que é *Records Variants*.

Temos um exemplo aqui bem simples, e o que são essas *Records Variants* e qual é o objetivo delas?

Antes de explicar a fundo sobre elas, vale a pena saber que são recursos disponíveis na linguagem.

Antigamente nós tínhamos dois problemas muito grande que era armazenamento e processamento.

E antigamente tínhamos pouquíssimos recursos de memórias, e gerenciar memória da forma mais produtiva possível era um diferencial para a linguagem.

As Records Variantes estão aí para isso, só para contextualizar vocês nisso.

Então vamos ver o que é *Records Variants*.

```
1 | TForma = record
2 |     Nome : String;
3 |     case isQuadrado : Booleana of
4 |         True : (X1, Y1, X2, Y2 : Integer);
5 |         False : (X, Y : Integer);
6 |     end;
```

# RECURSOS AVANÇADOS NO DELPHI

## {CONHECENDO AS RECORDS VARIANTS}

Observe que temos um *record* no código acima chamado **TForma**, e o que tenho nesse meu *record*, é o seguinte, o Nome é uma **string**, e tenho um **case of** dentro do record.

Dentro da estrutura do record colocamos um case, onde verifico se o **isQuadrado** é verdadeiro ou falso.

Se minha forma for verdadeira, eu irei ter os parâmetros X1, X2, Y2, Y1, ou seja, os quatro cantos do quadrado, e se não for, irei ter apenas os valores X e Y.

Já podemos perceber que essa minha *record* é uma variante, ou seja, uma record mutável.

Então se a forma for um quadrado, eu irei acessar X1, X2, Y2, Y1, e se eu tentar acessar X, o que irá acontecer?

Dá Erro? Não dá erro.

Se eu tentar acessar Y mesmo ela sendo um quadrado da erro?

Também não dá erro.

Então para que serve esse esse **case** aí no records?

Para contextualizar, isso está ligado ao gerenciamento de memória.

O que acontece, quando você coloca uma estrutura condicional dentro do record, o compilador interpreta o seguinte, se for um quadrado, vai ter X1, Y1, X2, Y2, porém X e Y, que seriam as opções de não ser um quadrado, elas ocupam o mesmo lugar na memória de X1 e Y1.

# RECURSOS AVANÇADOS NO DELPHI

## {CONHECENDO AS RECORDS VARIANTS}

Antigamente o gerenciamento de memória era vital, quando tínhamos, por exemplo, em um PC XT com 640KB de ram.

Mas hoje em dia, uma máquina simples já possui uns 6GB de ram, e isso não é perceptível, quando programamos de qualquer forma, pois temos memória ram de sobra.

Mas em máquinas antigas isso era primordial.

Deixa só uns vazamentos de memórias para ver o que acontece, ou não. Gerencie da melhor forma para ver o que aconteceria.

O que acontece é um aproveitamento, vamos dizer assim, de espaço na memória.

Quando colocamos esse case dentro do record, tanto o X1, quanto o X, seja ele verdadeiro ou falso, estão ocupando o mesmo endereço de memória.

Um exemplo que temos, na documentação do Delphi.

Temos uma classe *TFuncionario* que é um record, onde temos as propriedades Nome, que é uma string, e vamos dizer que o funcionário irá ter dois tipos de salário, onde tem funcionários que são remunerados por hora, ou remunerado mensalmente.

# RECURSOS AVANÇADOS NO DELPHI

## {CONHECENDO AS RECORDS VARIANTS}

```
1 | TFuncionario = record
2 |   Nome: String;
3 |   SalarioHora : Currency;
4 |   SalarioMensal : Currency;
5 | end;
```

Percebe que ao fazer esse procedimento estamos alocando dois espaços de memória, um para salário por hora, e um para salário por mês.

Porém um funcionário nunca vai ter os dois tipos de salário ao mesmo tempo.

Esse funcionário, ou recebe por hora, ou por mês.

Então para aproveitar a memória, colocamos um case.

```
1 | TFuncionario = record
2 |   Nome: String;
3 |   case Contrato : Boolean of
4 |     True : (SalarioMensal : Currency);
5 |     False : (SalarioHora : Currency);
6 | end;
```

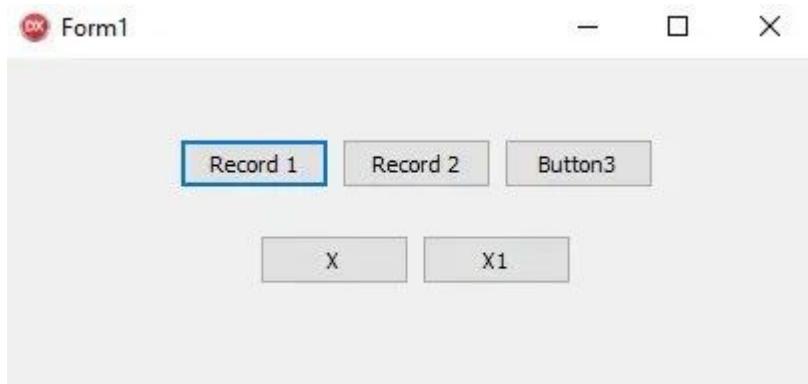
Verificamos se o funcionário é contratado ele tem o salarioMensal, se não for, SalarioHora.

Como essa condição nunca aconteceria, as duas ao mesmo tempo, acabamos aproveitando esse espaço de memória, onde em um único lugar da memória, estará salvo, seja o salário por hora, ou salário mensal.

# RECURSOS AVANÇADOS NO DELPHI

## {CONHECENDO AS RECORDS VARIANTS}

Usamos a opção de contrato para recuperar o que quiser.



Um exemplo claro disso, é a implementação que fizemos no evento de click do botão Record 1.

```
1 procedure TForm1.ButtonClick(Sender : TObject);
2 begin
3     Quadrado.Nome := 'Meu Quadrado';
4     Quadrado.isQuadrado := True;
5     Quadrado.X1 := 10;
6     Quadrado.X2 := 20;
7     Quadrado.Y1 := 30;
8     Quadrado.Y2 := 40;
9 end;
```

Nesse exemplo eu passo um nome para o quadrado, e digo que é um quadrado, e passo as informações do quadrado.

E no botão X, eu estou recuperando o valor de X, e no botão X1, eu recupero o valor do X1.



# RECURSOS AVANÇADOS NO DELPHI

## {CONHECENDO AS RECORDS VARIANTS}

Observe que tanto o X quanto X1 tem o mesmo valor, porque os dois estão no mesmo lugar da memória, ou seja, estamos otimizando o consumo de memória, para que o nosso software não aloque recursos da memória desnecessariamente.

Esse é o conceito das *Records Variante*.

Você pode criar atributos condicionais para ter diversos campos alocados no mesmo lugar da memória, porém, como esses campos são únicos, individuais, ou seja, depende, da condicional que estiver tratando.

Como na classe de funcionários, ou ele é contratado ou não, ele não tem como ser os dois, então você utilizar de um mesmo campo para buscar essa informação e utilizá do atributo condicional booleano para saber que tipo de salário ele irá receber.

É muito simples, e fácil, mas é uma funcionalidade que fica aqui para seu conhecimento.

Como já falei, nem tudo que vemos, iremos colocar em produção em nossos softwares, mas é um conhecimento que irá contribuir para você na sua caminha na programação.

**RECURSOS  
AVANÇADOS  
NO DELPHI**



**{INTRODUZINDO  
VARIÁVEIS  
INLINE}**

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUZINDO VARIÁVEIS INLINE}

À partir da versão 10.3 do Delphi, é possível utilizar um recurso muito útil para a linguagem, variáveis inline locais com escopo local e inferência de tipos.

A linguagem Delphi em 10.3 tem uma mudança bastante central na maneira como permite muito mais flexibilidade na declaração de variáveis locais, seu escopo e tempo de vida. Esta é uma mudança que quebra um princípio fundamental da linguagem Pascal original, mas oferece um número significativo de vantagens, reduzindo o código desnecessário em vários casos.

### Estilo antigo do bloco var.

Desde o Turbo Pascal 1 e até agora, seguindo as regras clássicas da linguagem Pascal, todas as declarações de variáveis locais tinham que ser feitas em um bloco var escrito antes do início dos métodos:

```
1 | procedure Teste;
2 | var
3 |   I: integer;
4 | begin
5 |   I:= 22;
6 |   ShowMessage(I.ToString());
7 | end;
```

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUCINDO VARIÁVEIS INLINE}

### Declarações variáveis Inline

A nova sintaxe de declaração de variável inline permite que você declare a variável diretamente em um bloco de código (permitindo também vários símbolos como de costume):

```
1 procedure Test;
2 begin
3     var I, J: Integer;
4     I := 22;
5     j := I + 20;
6     ShowMessage (J.ToString);
7 end;
```

Embora isso possa parecer uma diferença limitada, há vários efeitos colaterais dessa alteração. São esses efeitos adicionais que tornam o recurso valioso.

### Inicializando Variáveis Inline

Uma mudança significativa em comparação com o modelo antigo é que a declaração e inicialização inline de uma variável podem ser feitas em uma única instrução. Isso torna as coisas mais legíveis e mais suaves em comparação com a inicialização de várias variáveis no início de uma função.

```
1 procedure Test;
2 begin
3     var I: Integer := 22;
4     ShowMessage (I.ToString);
5 end;
```

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUZINDO VARIÁVEIS INLINE}

Mais ainda, se o valor de uma variável estiver disponível apenas mais tarde no bloco de código, em vez de definir um valor inicial (como 0 ou nil) e posteriormente atribuir o valor real, você pode atrasar a declaração de variável ao ponto em que você pode calcular bom valor inicial:

```
1 procedure Test1;
2 begin
3     var I: Integer := 22;
4     var J: Integer := 22 + I;
5     var K: Integer := I + J;
6     ShowMessage (K.ToString());
7 end;
```

Em outras palavras, enquanto no passado todas as variáveis locais eram visíveis em todo o bloco de código, agora uma variável inline é visível apenas da posição de sua declaração e até o final do bloco de código.

## Escopo e tempo de vida de variáveis inline declaradas em blocos aninhados

A limitação de escopo também é mais relevante, pois não se aplica a todo o método, mas somente ao bloco de código de begin-end a variável inline é exibida. Em outras palavras, o escopo de uma variável inline é limitado ao bloco de declaração e a variável não pode ser usada fora do bloco. Não apenas, mas a vida útil variável também é limitada ao bloco. Um tipo de dados gerenciado (como uma interface, uma string ou um registro gerenciado) será descartado no final do sub-bloco, em vez de invariavelmente no final do procedimento ou método.

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUCINDO VARIÁVEIS INLINE}

```
1 procedure Test2;
2 begin
3     var I: Integer := 22;
4     if I > 10 then
5         begin
6             var J: Integer := 3;
7             ShowMessage (J.ToString);
8         end
9     else
10        begin
11            var K: Integer := 3;
12            ShowMessage (J.ToString); // COMPILER ERROR: "Identificador não declarado: J"
13        end;
14    // J e K não acessíveis aqui no
15 end;
```

Como você pode ver no último snippet de código acima, uma variável declarada dentro de um bloco begin-end é visível apenas no bloco específico, e não depois que o bloco foi finalizado. No final das instruções if, J e K não serão mais visíveis.

Como mencionei, o efeito não se limita apenas à visibilidade. Uma variável gerenciada, como uma referência de interface ou um registro, será limpa adequadamente no final do bloco, em vez de no final do procedimento ou método:

```
1 procedure Test99;
2 begin
3     if (something) then
4         begin
5             var Intf: IInterface = GetInterface;
6             var MRec: TManagedRecord = GetMRecValue;
7             UseIntf(Intf);
8             UseMRec(MRec);
9         end;
10    end;
```

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUCINDO VARIÁVEIS INLINE}

### Inferência de tipo para variáveis inline

Outro grande benefício de variáveis inline é que o compilador agora pode, em várias circunstâncias, inferir o tipo de uma variável inline observando o tipo da expressão ou valor atribuído a ela:

```
1 procedure Test;
2 begin
3     var I := 22;
4     ShowMessage (I.ToString);
5 end;
```

O tipo da expressão de r-valor (isto é, o que vem depois de `:=`) é analisado para determinar o tipo da variável. Alguns dos tipos de dados são “expandidos” para um tipo maior, como no caso acima, onde o valor numérico 22 (um `ShortInt`) é expandido para `Integer`. Como regra geral, se o tipo de expressão à direita for um tipo integral e menor que 32 bits, a variável será declarada como um inteiro de 32 bits. Você pode usar um tipo explícito se quiser um tipo numérico específico e menor.

Agora, embora esse recurso possa economizar alguns toques no teclado para um `Integer` ou uma `string`, a inferência de tipos de variáveis se torna bastante interessante no caso de tipos complexos, como instâncias de tipos genéricos. No trecho de código abaixo, os tipos inferidos são `TDictionary` para a variável `MyDictionary` e `TPair` para a variável `apair`.

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUCINDO VARIÁVEIS INLINE}

```
1 procedure NewTest;
2 begin
3     var MyDictionary := TDictionary<string, Integer>.Create
4     MyDictionary.Add('one', 1);
5     var APair := MyDictionary.ExtractPair('one');
6     ShowMessage(APair.Value.ToString)
7 end;
```

## Constantes Inline

Além das variáveis, agora você também pode inserir uma declaração de valor constante. Isso pode ser aplicado a constantes de tipos ou constantes não tipadas, caso em que o tipo é inferido (um recurso que está disponível para constantes há muito tempo). Um exemplo simples está abaixo:

```
1 const M: Integer = (L + H) div 2;
2 const M = (L + H) div 2;
```

## Loops for para declaração de variável de loop in-line

Outra circunstância específica na qual você pode tirar proveito das declarações de variáveis inline é com as instruções loop for, incluindo os loops for-to clássicos e os loops for-in modernos:

```
1 for var I: Integer := 1 to 10 do ...
2 for var Item: TItemType in Collection do...
```

Você pode simplificar ainda mais o código aproveitando a inferência de tipos:

```
1 for var I := 1 to 10 do ...
2 for var Item in Collection do ...
```

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUZINDO VARIÁVEIS INLINE}

Este é um caso em que ter a variável inline com escopo limitado é particularmente benéfico, como no exemplo de código abaixo:

Usar a variável `I` fora do loop causará um erro do compilador (enquanto era apenas um aviso na maioria dos casos no passado):

```
1 procedure ForTest;
2 begin
3   var total := 0;
4   for var I: Integer := 1 to 10 do
5     Inc (Total, I);
6   ShowMessage (total.ToString());
7   ShowMessage (I.ToString());
8 end;
```

## Resumo Inline

Declarações de variáveis inline, com inferência de tipos e escopo local, trazem ar fresco para a linguagem Delphi. Embora a legibilidade do código-fonte do Pascal continue sendo um princípio fundamental a ser preservado, é importante modernizar a linguagem no nível principal, removendo um pouco da ferrugem (real ou percebida). No caso de variáveis inline, há muitas vantagens de sair do estilo tradicional de codificação, que o valor é claro.

Ainda assim, você pode manter seu código como está e pedir a seus colegas desenvolvedores que sigam a declaração tradicional. Nada no código ou estilo existente está errado, mas as **declarações inline** oferecem uma nova oportunidade. Eu já tenho problemas para voltar para versões mais antigas do Delphi.

**RECURSOS  
AVANÇADOS  
NO DELPHI**



**MultiPaste  
no IDE do  
RAD Studio**



# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUCINDO VARIÁVEIS INLINE}

Sejamos sinceros,

Quantos de nós programadores Delphi já realizamos operações entediantes como concatenar strings extensas, como por exemplo uma query, onde devemos realizar uma operação em tempo de execução e precisamos transformar essa query em uma string para colocarmos numa propriedade SQL da consulta.

Neste capítulo, irei lhe mostrar algo que me ajuda muito em muitos, vou lhe mostrar o “**MultiPaste**”.

### O que é MultiPaste?

Talvez você esteja se perguntando o que é esse tal de **MultiPaste**, e por anos de experiência na ferramenta RAD da Embarcadero nunca ouviu falar nessa funcionalidade maravilhosa que está disponível no Delphi.

Já teve algum problema assim?

Você tem algum SQL:

```
1 | SELECT Customers.CustomerName, Orders.OrderID  
2 | FROM Customers  
3 | FULL OUTER JOIN Orders  
4 | ON Customers.CustomerID = Orders.CustomerID  
5 | ORDER BY Customers.CustomerName;
```

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUZINDO VARIÁVEIS INLINE}

e você deseja adicionar esse SQL à propriedade SQL de uma consulta em tempo de execução. Você acaba tendo que transformar isso em uma string como esta:

```
1 | 'SELECT Customers.CustomerName, Orders.OrderID' +
2 | 'FROM Customers' +
3 | 'FULL OUTER JOIN Orders' +
4 | 'ON Customers.CustomerID = Orders.CustomerID' +
5 | 'ORDER BY Customers.CustomerName;'
```

ou manualmente em cada linha como esta:

```
1 | FDQuery1.SQL.Add('SELECT Customers.CustomerName, Orders.OrderID');
2 | FDQuery1.SQL.Add('FROM Customers');
3 | FDQuery1.SQL.Add('FULL OUTER JOIN Orders');
4 | FDQuery1.SQL.Add('ON Customers.CustomerID = Orders.CustomerID');
5 | FDQuery1.SQL.Add('ORDER BY Customers.CustomerName');
```

Ambas as escolhas são uma grande dor no código para o código.

Agora não mais!  
**MultiPaste** torna esse tipo de coisa pateticamente fácil!

Vamos escolher a segunda opção acima – a adição manual do código SQL uma linha de cada vez.

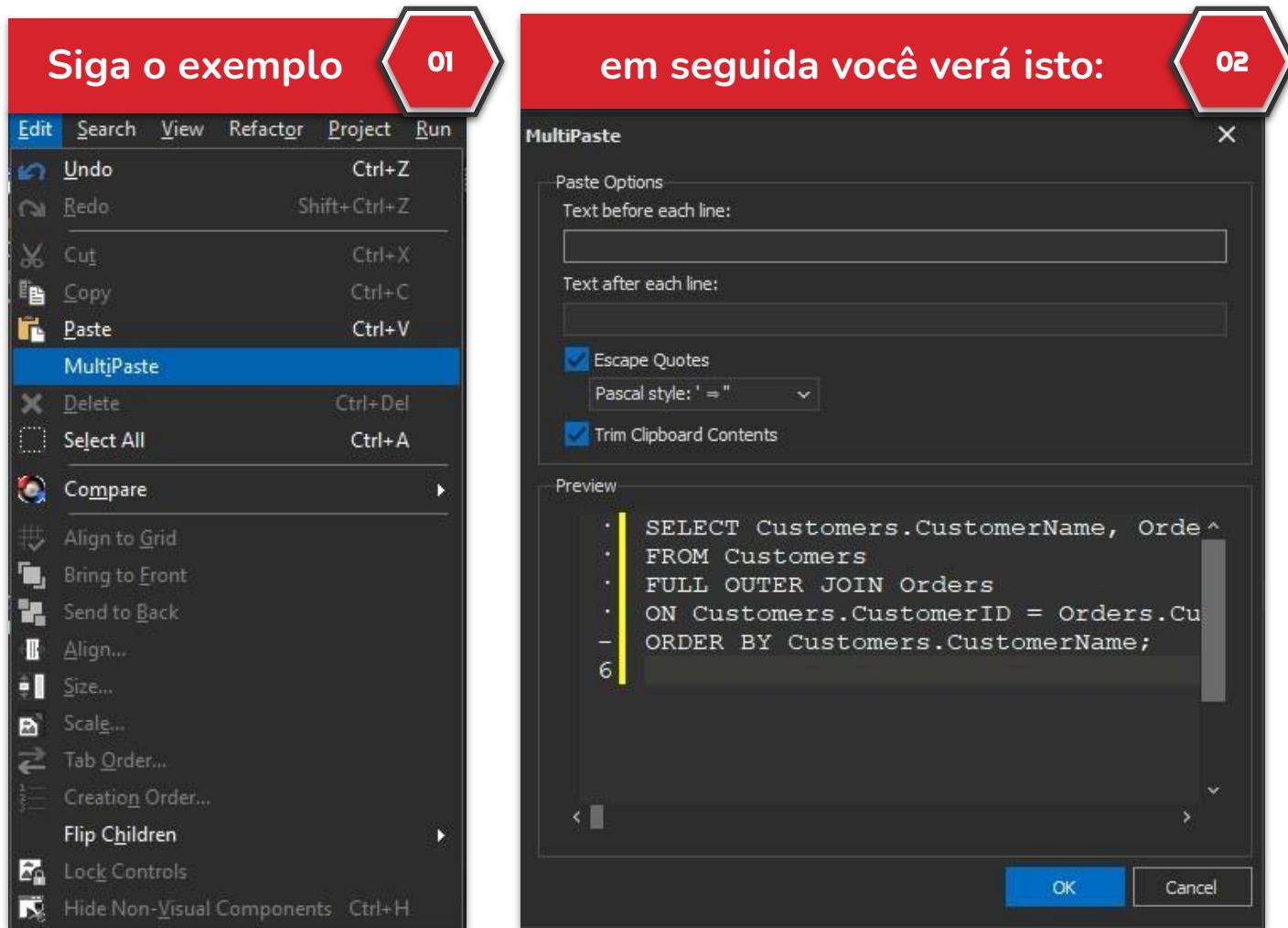
Primeiro, copie o SQL para sua área de transferência. Isso é importante – o texto que você deseja manipular deve estar em sua área de transferência.

Em seguida, coloque o cursor onde você deseja inserir o texto resultante.

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUCINDO VARIÁVEIS INLINE}

Em seguida, selecione **Editar | MultiPaste** no menu do IDE (você precisará ter uma janela de código aberta para que o item de menu esteja ativo).



Agora, a partir daí, você pode digitar isso na primeira caixa de edição:

Text before each line:  
FDQuery1.SQL.Add();

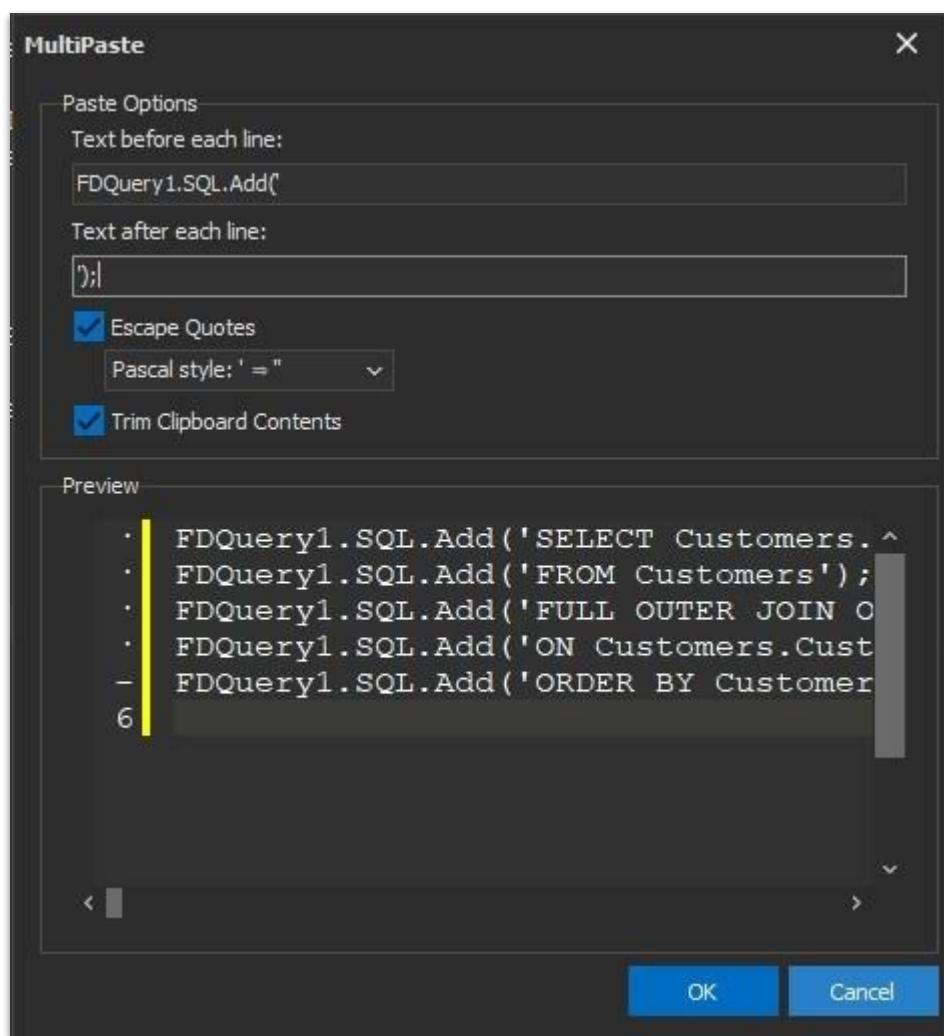
e na segunda caixa de edição digite:

Text after each line:  
);

# RECURSOS AVANÇADOS NO DELPHI

## {INTRODUCINDO VARIÁVEIS INLINE}

Você deve então observar que **memo** principal está alterando o texto que você tem em sua área de transferência adicionando o conteúdo da primeira caixa de edição ao início de cada linha e o conteúdo da segunda caixa de edição ao final de cada linha. Você deve acabar com uma caixa de diálogo que se parece com isso:



Em seguida, pressione Ok e o texto que você criou é inserido no ponto do cursor.

Assim, **MultiPaste** permite que você evite um pouco da codificação demorada e entediante que todos nós fizemos em momento ou outro.

**RECURSOS  
AVANÇADOS  
NO DELPHI**



**VCL E FMX  
NO MESMO  
PROJETO**

# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

Você já tentou implementar algo em seu projeto VCL que só tinha no Firemonkey?

Modernizar seu software é algo de extrema importância, seguir as boas práticas de UI/UX irá trazer muito mais clientes e os que já estão contigo fidelizar mais ainda.

Mas você deve estar se perguntando, será que terei que fazer um novo projeto para utilizar o Firemonkey?

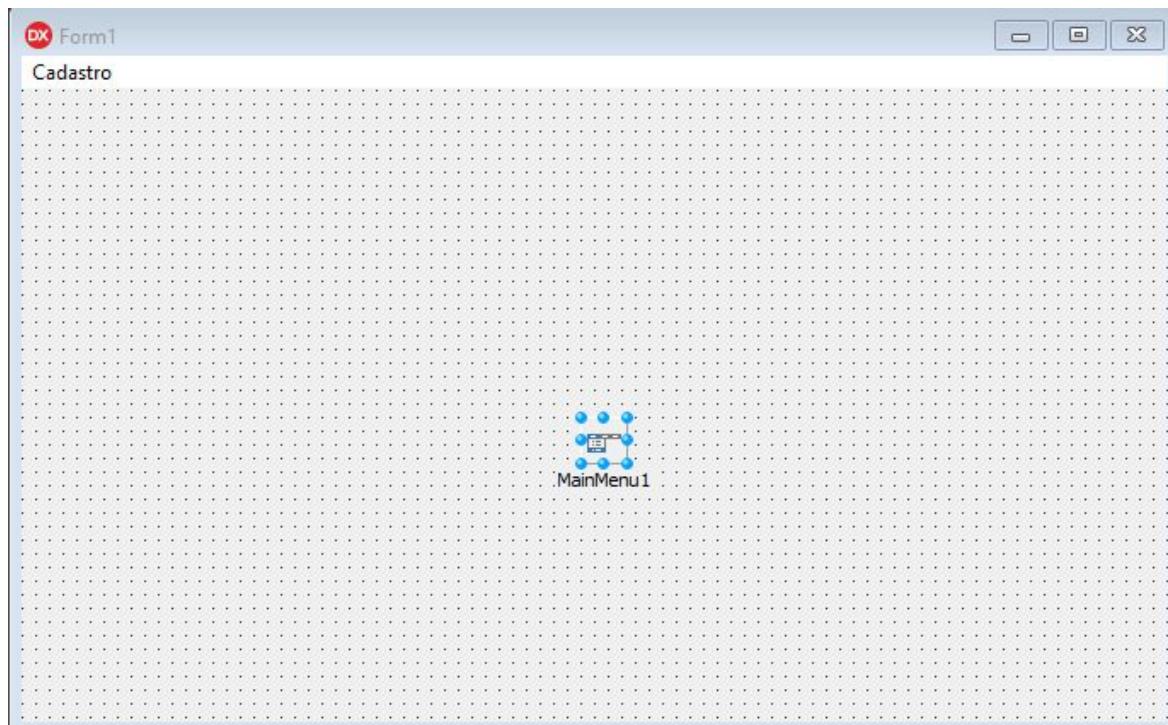
Te digo que você não terá necessidade disso, você pode integrar o que existe de melhor no FMX junto com o VCL.

Para quebrar esse paradigma, neste capítulo irei lhe ajudar.

Vou te mostrar como integrar num projeto VCL o FMX.

## Vamos lá então?

Eu posso um projeto VCL criado, com o componente MainMenu.

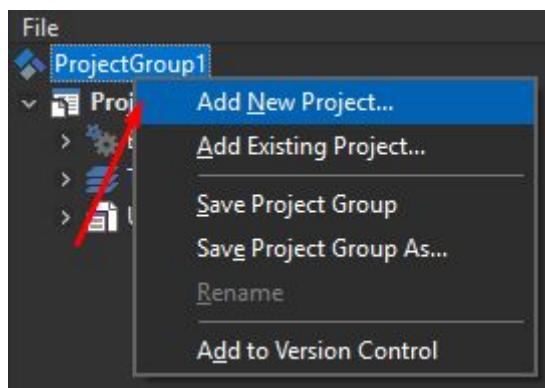


# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

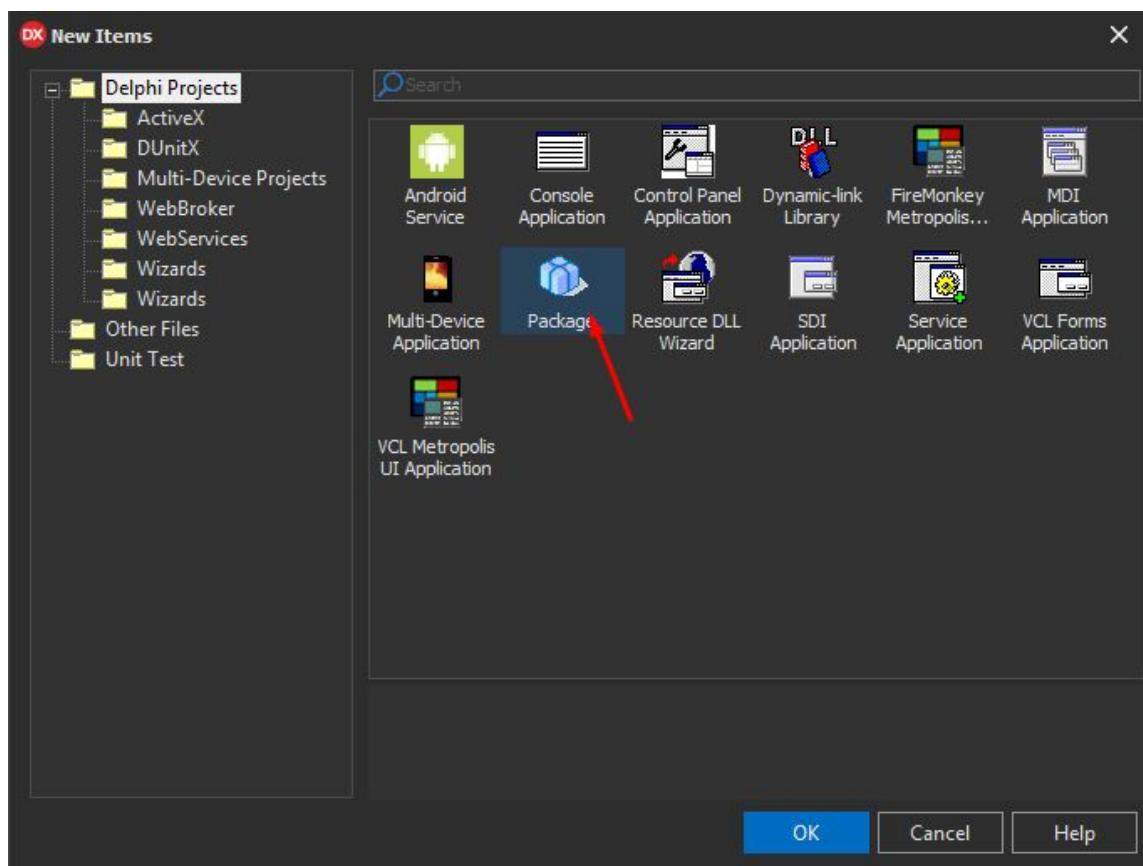
Agora esse menu de produtos desejo colocar dentro o formulário do FireMonkey.

## Mas como iremos fazer isso?

Muito simples, dentro do *ProjectGroup* vamos criar um novo projeto.

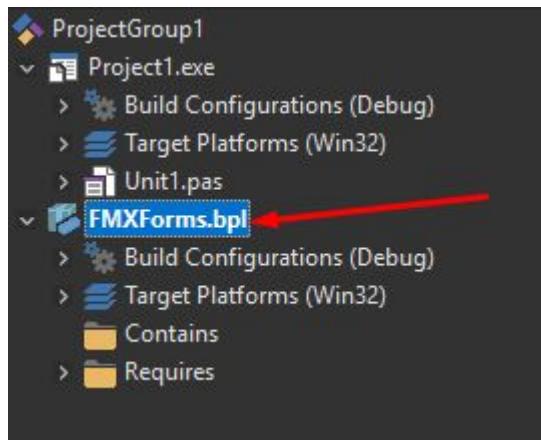


E vamos criar um novo *Package*.

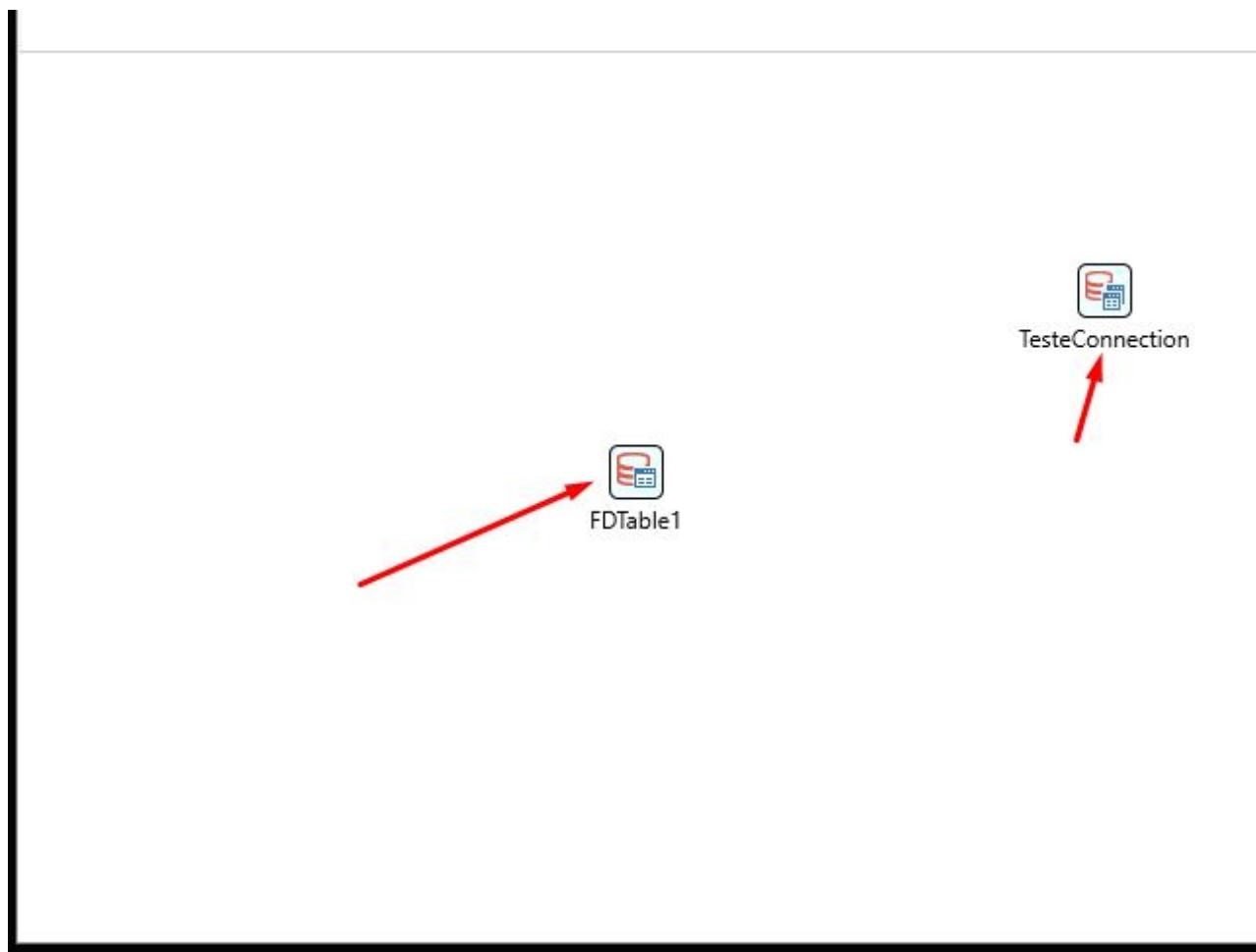


# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

Vamos deixar esse nosso Package com o nome de *FMXForms*.

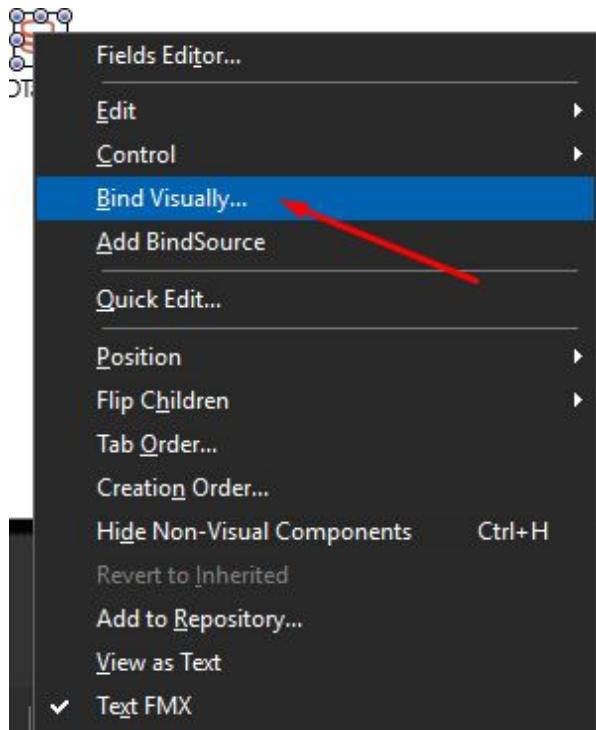


Dentro desse nosso Package iremos adicionar um *Mult-Device Form*, neste form iremos colocar um *GRID* ao *Client*, um *FDConnection* e um *FDTTable*.

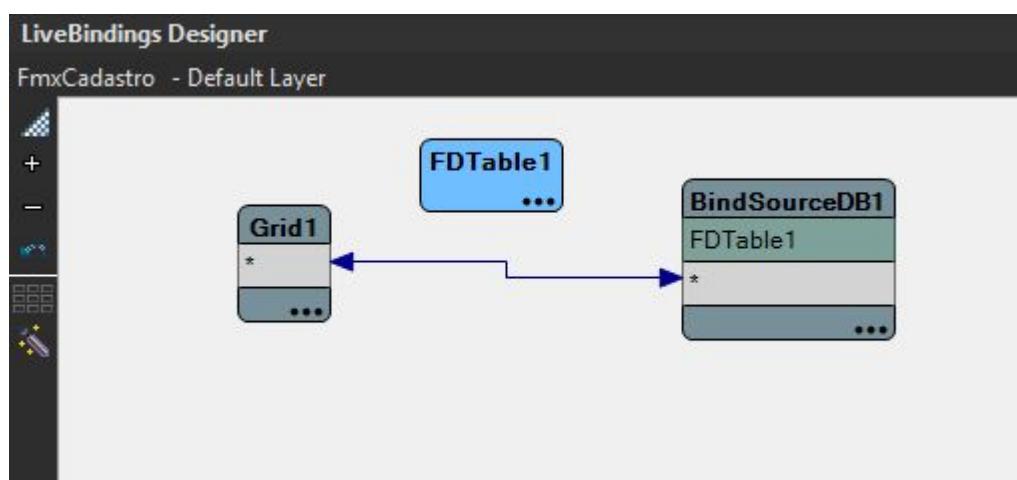


# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

Vamos abrir o *Bind Visually* do Firemonkey para fazermos a conexão dos componentes.

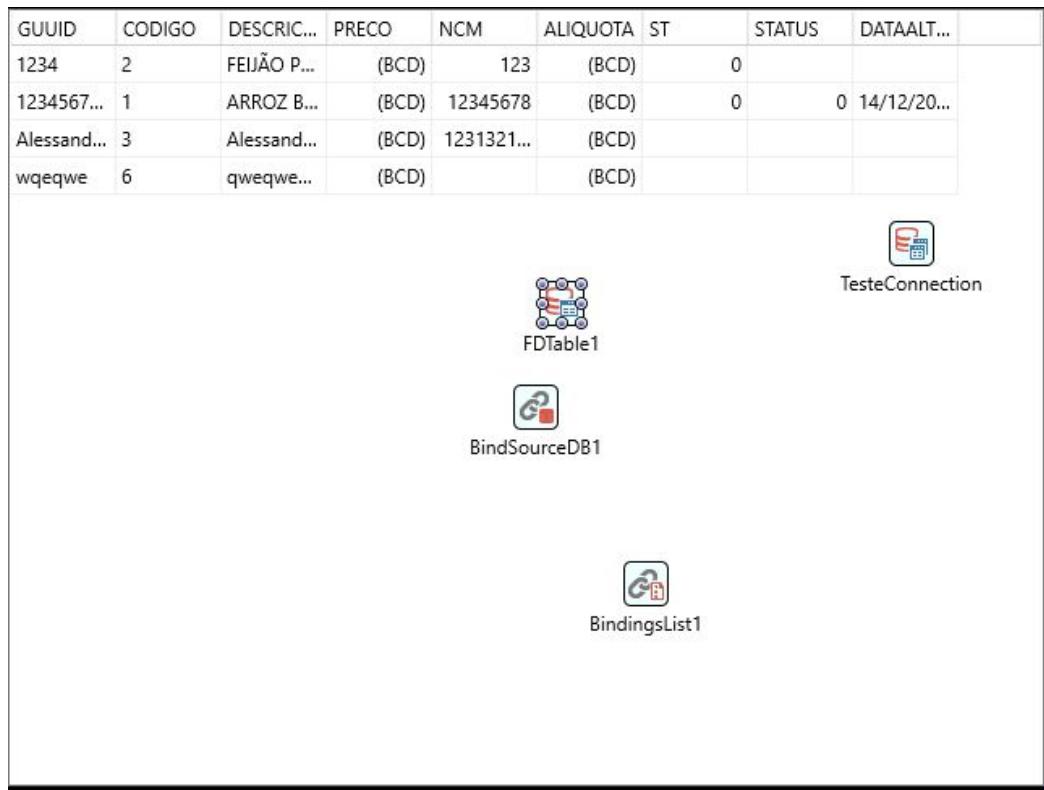


Fazemos as ligações no *Bind* do *FDTable1* com o *Grid*.

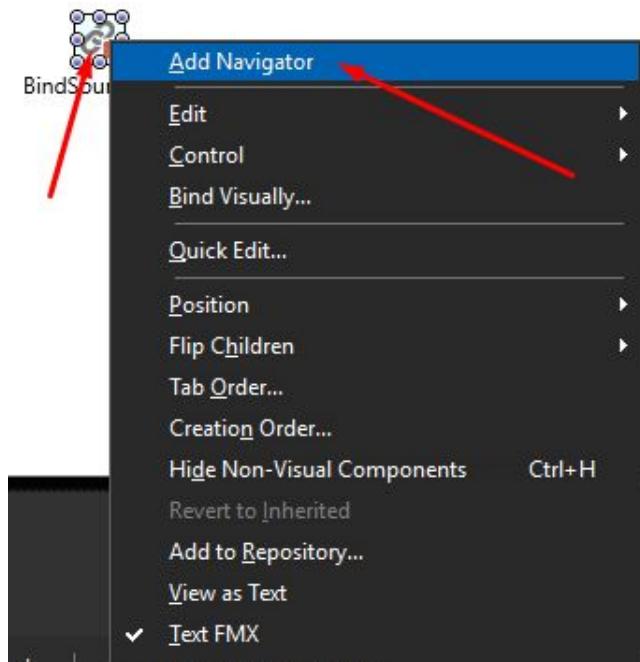


# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

Feito isso se mandarmos ativar nosso *FDTable* já teremos todos os dados.

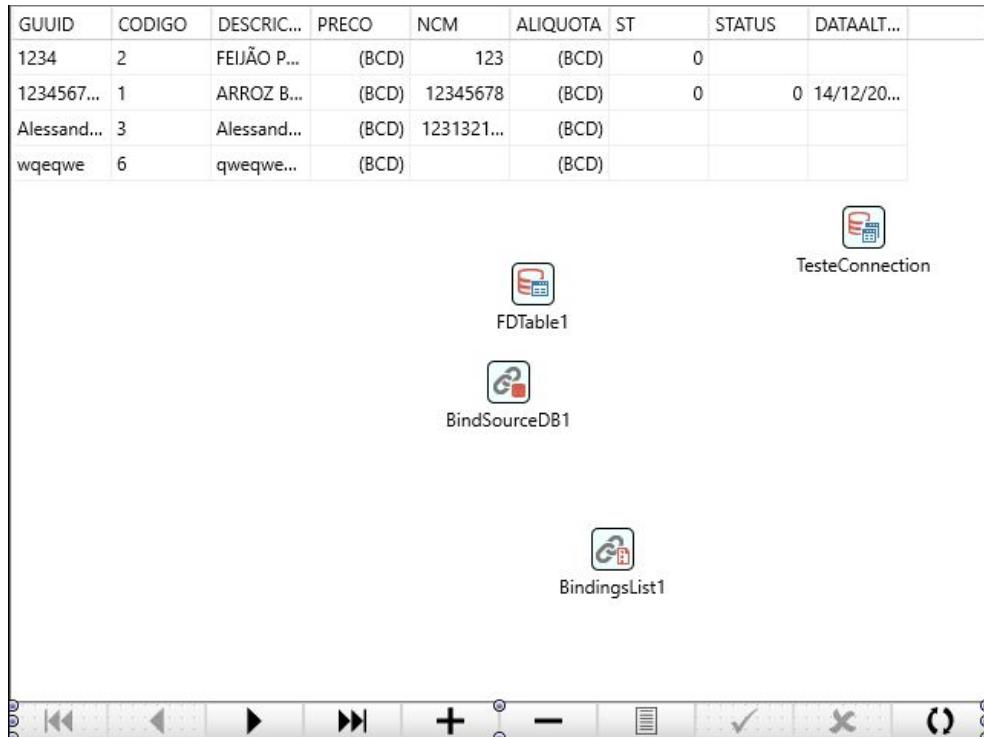


Agora vou com o botão direito em *BindSourceDB1* e irei adicionar um *Navigator*, muito semelhante com o que usamos no VCL.

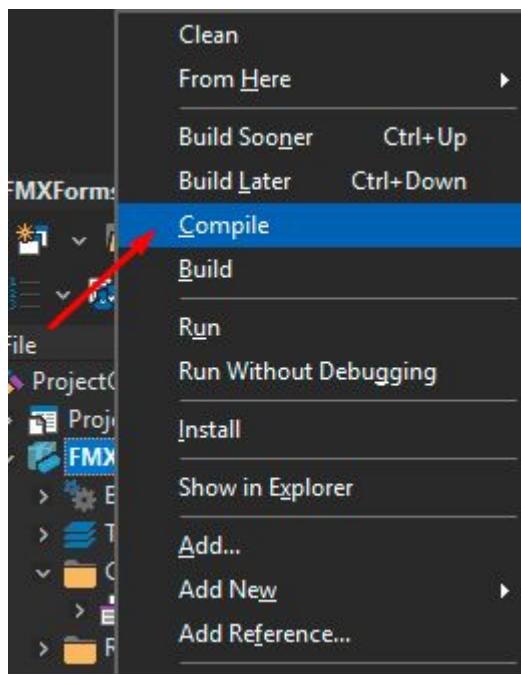


# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

Nós já temos um form e listando os dados.



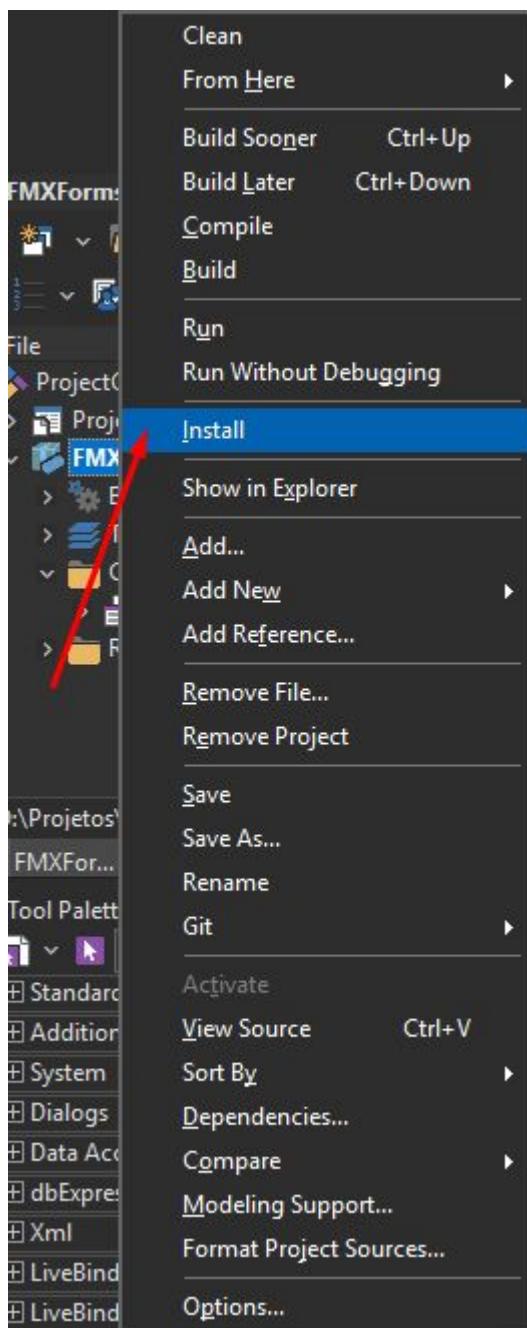
Agora mandamos compilar esse Package que acabamos de criar.



# RECURSOS AVANÇADOS NO DELPHI

## {VCL E FMX NO MESMO PROJETO}

E vamos instalar agora.



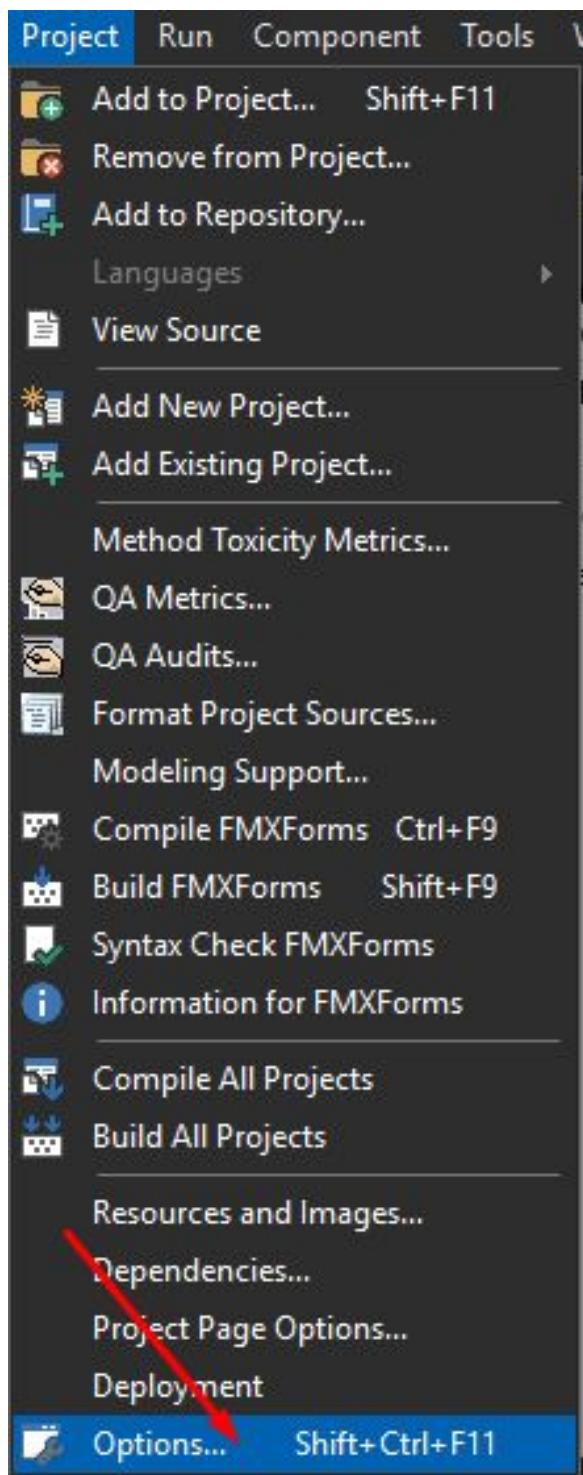
Agora temos nosso projeto instalado perfeitamente.

# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

## E agora?

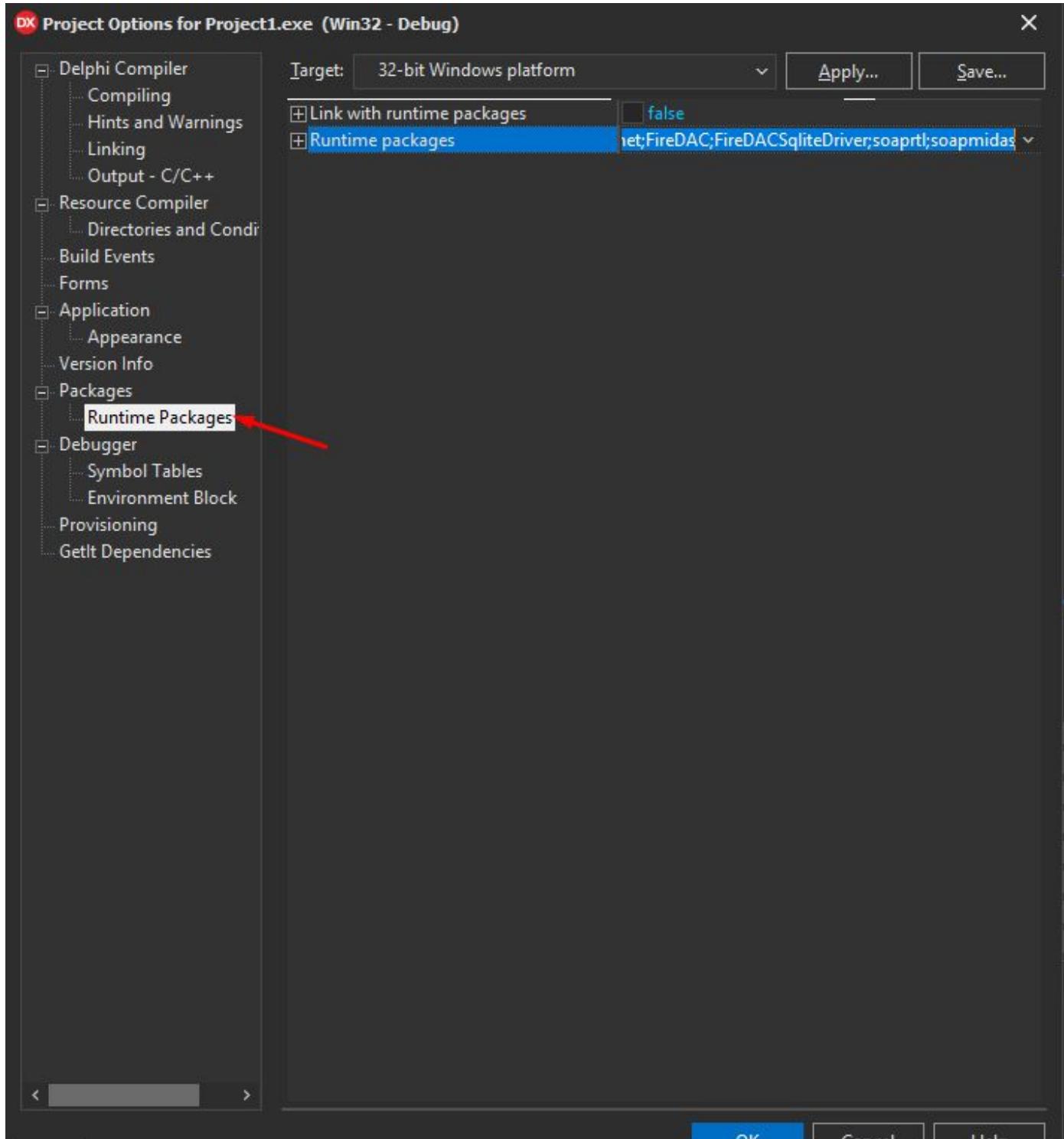
Agora vamos ao nosso projeto *VCL*, e iremos fazer algumas coisas para que esse nosso formulário *Firemonkey* seja reconhecido.

Vamos no menu *Project > Options*.



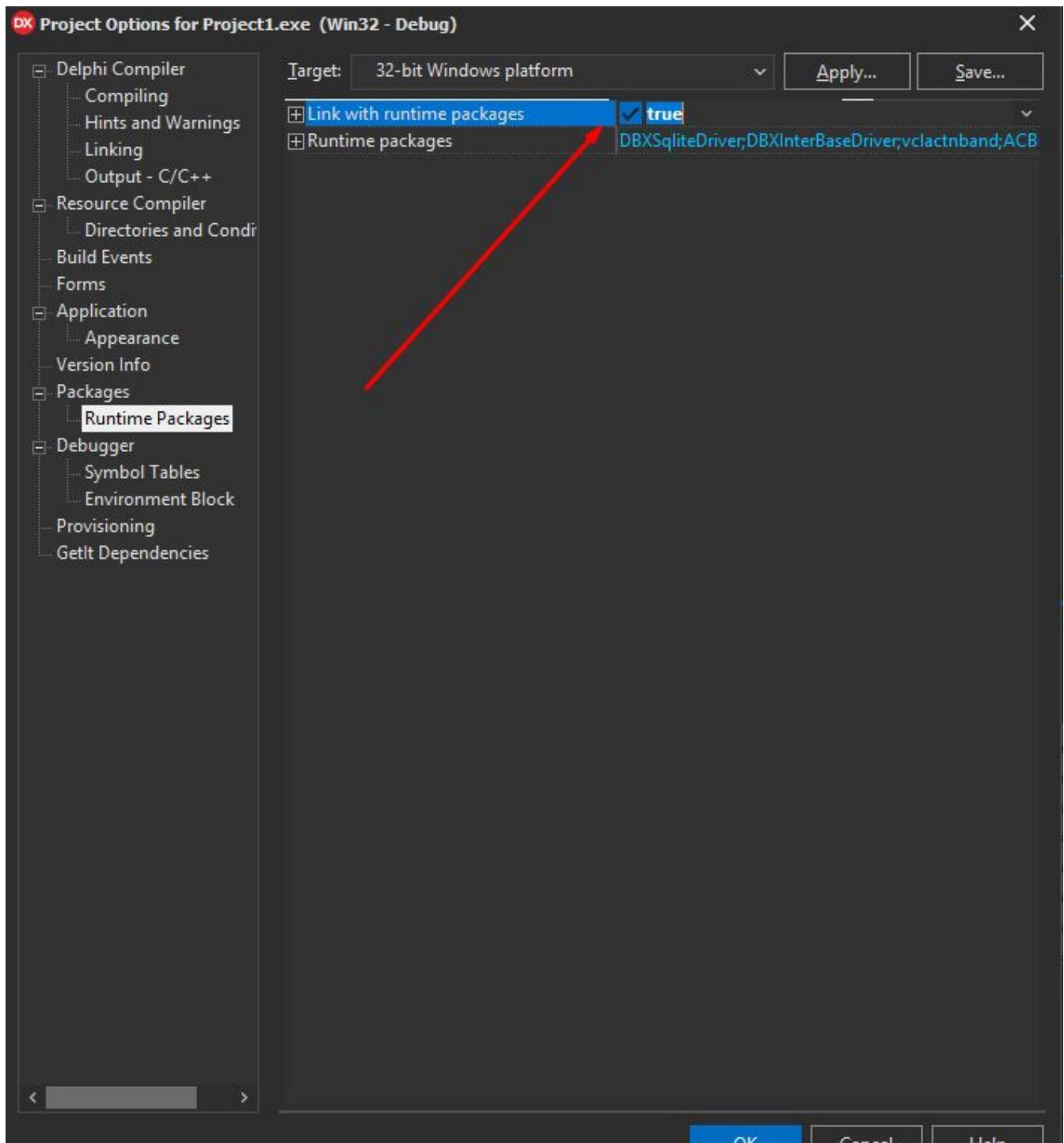
# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

E na tela que segue vamos na opção Package > Runtime Packages.



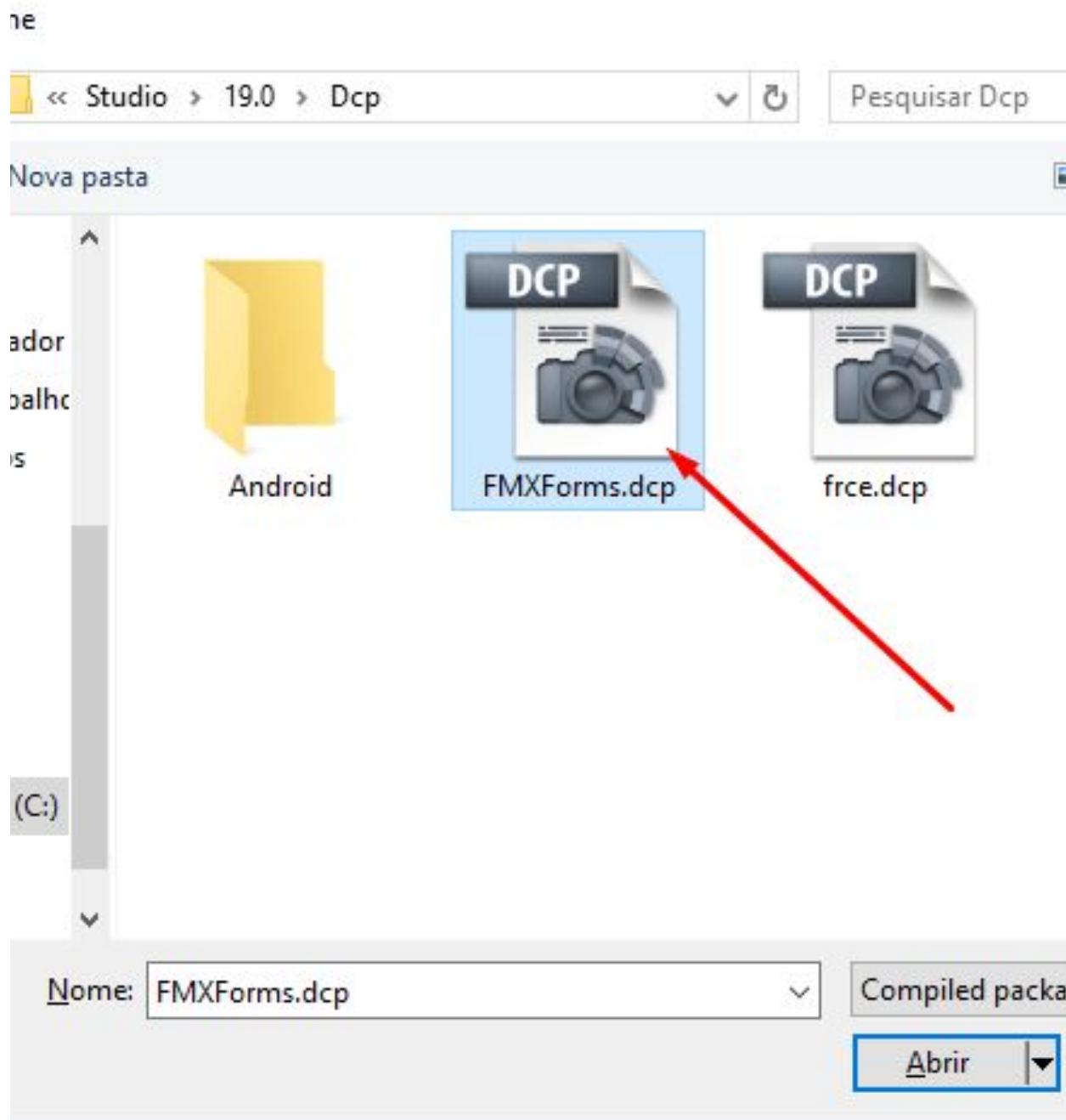
# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

E devemos marcar a opção *Link with runtime packages* deixando-a como *True*.



# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

E na opção Runtime Packages devemos procurar nosso pacote que acabamos de criar, geralmente ele fica em C:\Usuários\Público\Documentos Públcos\Embarcadero\Studio\<versão do seu delphi>\Dcp dentro dessa pasta encontramos o nosso pacote Dcp criado.



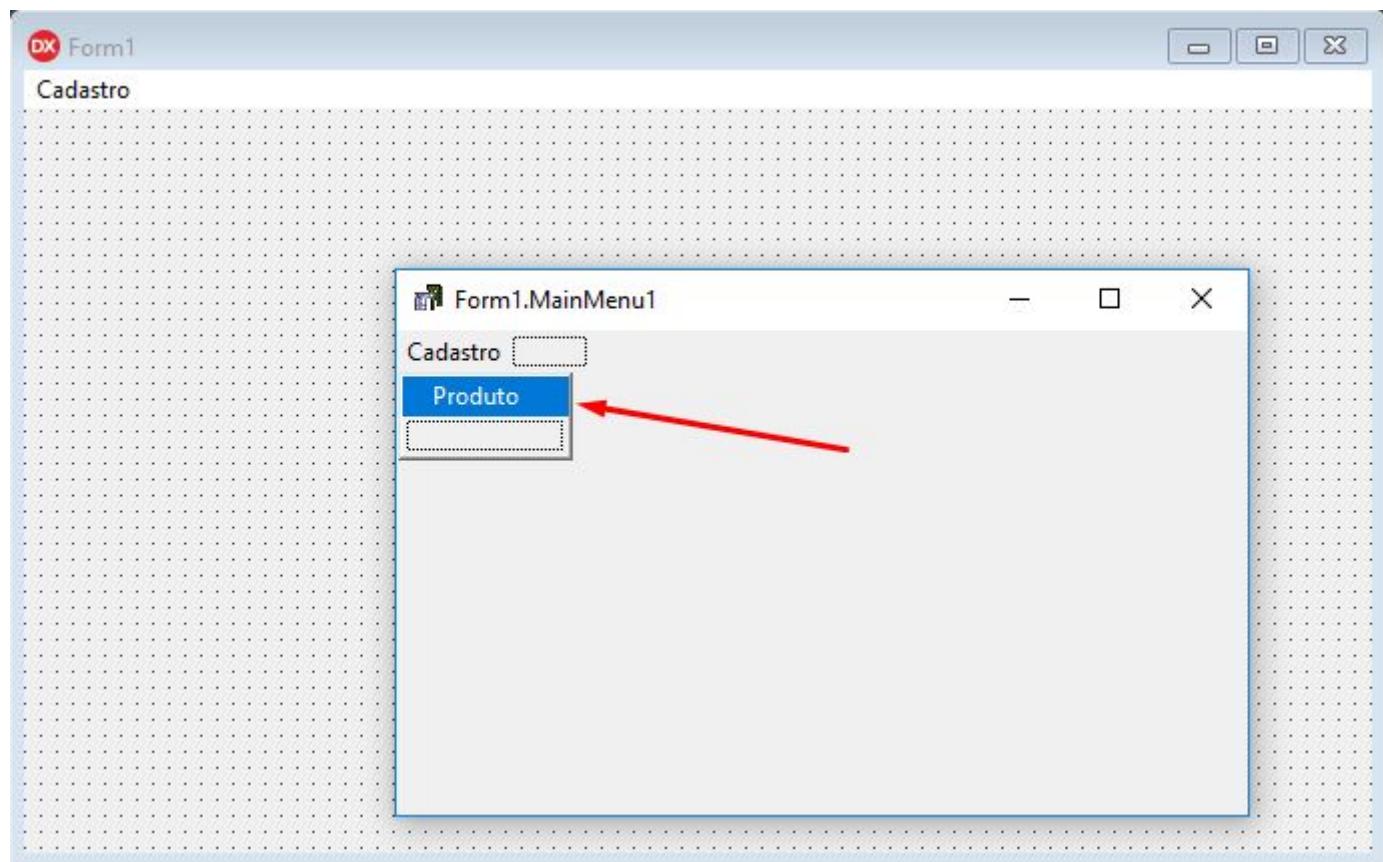
# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

Agora já temos esse nosso componente incluso em nosso projeto VCL.

**E agora como podemos vincular os dois?**

Devemos declarar as `Uses` do nosso componente no nosso projeto VCL.

E no menu que criamos no nosso projeto VCL em Produtos iremos chamar nosso formulário FMX.



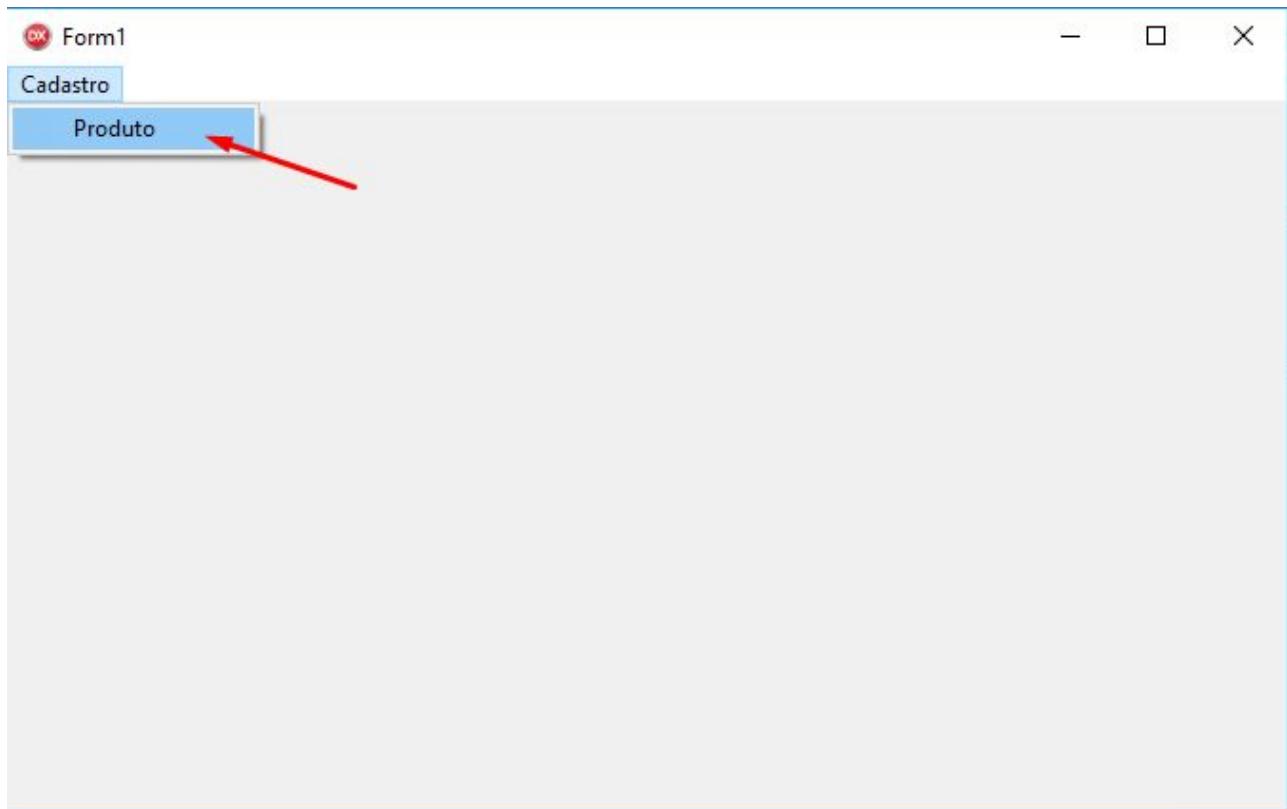
# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

Onde no código iremos deixar desta forma, seguimos o padrão de criação de formulários.

```
1 procedure TForm1.Produto1Click(Sender: TObject);
2 var
3   FMXForm : TFmxCadastro;
4 begin
5   FMXForm := TFmxCadastro.Create(Self);
6   try
7     FMXForm.ShowModal;
8   finally
9     FMXForm.Free;
10  end;
11 end;
```

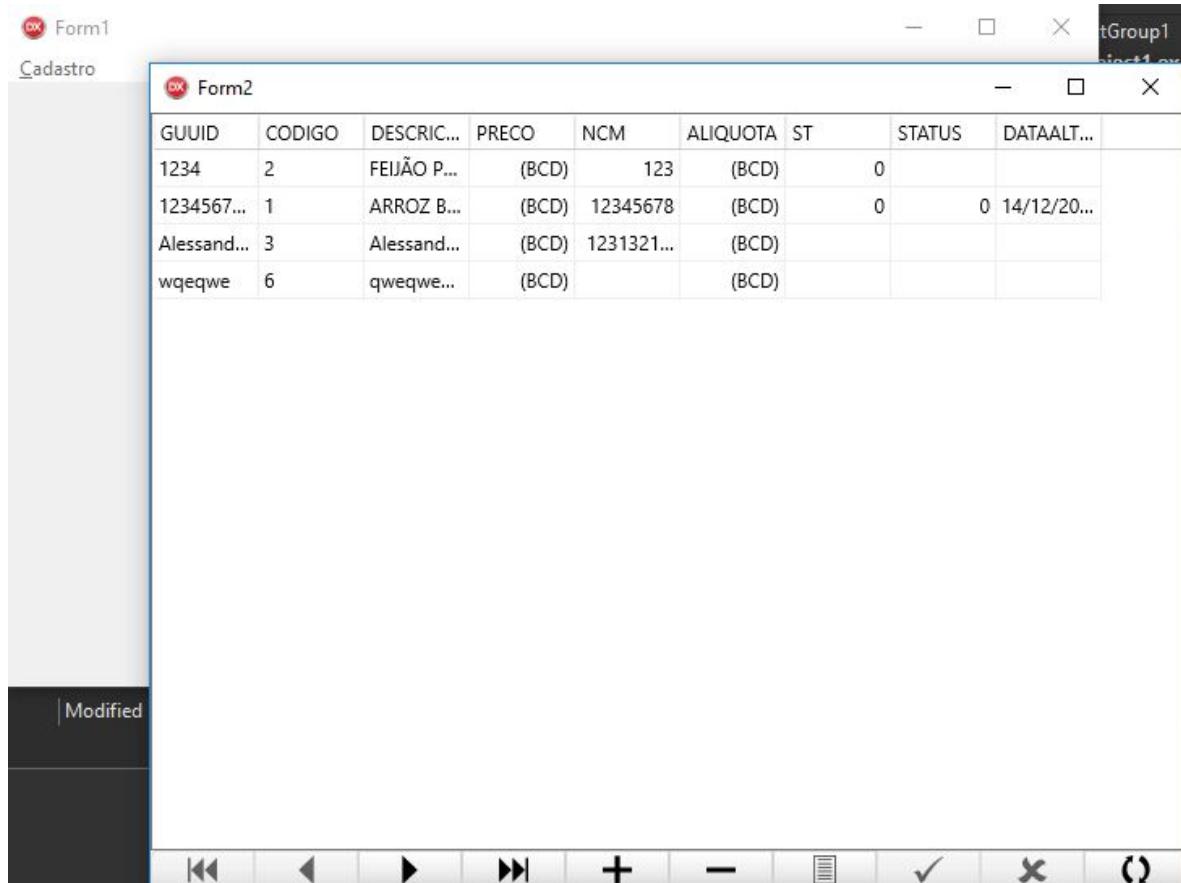
Feito isso, compilamos e executamos nosso projeto VCL.

Vamos no menu *Cadastro > Produto*.



# RECURSOS AVANÇADOS NO DELPHI {VCL E FMX NO MESMO PROJETO}

E ao clicarmos veja nosso formulário *FMX* funcionando juntamente com o nosso projeto *VCL*.



Viu como de forma prática e simples temos o melhor dos dois mundos no nosso Delphi.

**RECURSOS  
AVANÇADOS  
NO DELPHI**



**{CLASS  
HELPER}**



# RECURSOS AVANÇADOS NO DELPHI

## {CLASS HELPER}

```
while not dmDados.QueryUsuarios.Eof do
begin
  Operador := TOperador.Create;
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```

É muito comum durante o desenvolvimento a necessidade de adicionarmos novas funcionalidades, a métodos já existentes no Delphi, e em seus componentes, como por exemplo a validação de e formatação de dados.

Muito optam a trabalhar com herança, ou até mesmo a alterar diretamente no fonte do componente, e essas não são boas práticas.

Desde a versão 2006 do Delphi foi adicionado o suporte a Class helper, dando a possibilidade de adicionarmos novos comportamentos aos componentes.

## O que é class helper?

Uma classe ou um *record helper* é um tipo que – quando associado a outra classe ou registro – introduz nomes de métodos e propriedades adicionais que podem ser usados no contexto do tipo associado (ou seus descendentes). Os *helpers* são uma maneira de estender uma classe sem usar herança, o que também é útil para registros que não permitem a herança. Um *helper* simplesmente introduz um escopo mais amplo para o compilador usar ao resolver identificadores. Quando você declara uma classe ou um *record helper*, você declara o nome do *helper* e o nome do tipo que vai estender com o *helper*. Você pode usar o *helper* em qualquer lugar onde possa usar legalmente a classe ou o *record*. O escopo de resolução do compilador então se torna o tipo original, mais o auxiliar.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS HELPER}

```
while not dmDados.QueryUsuarios.EOF do
begin
  Operador := TOperador(dmDados.QueryUsuarios);
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```

Outro diferencial é que Class Helpers não se aplicam somente a componentes. Eles também permitem modificar classes de domínio criadas pelo desenvolvedor, adicionar métodos a estruturas do tipo record e tipos primitivos. Nesse último caso, no entanto, não é possível adicionar propriedades.

Neste capítulo, vamos falar mais do incômodo. Uma coisa que tem que despertar em você como desenvolvedor é o incomodo quando o código não está legal, o código está feio, não está legível, isso tem que começar a te incomodar.

### Observe o código abaixo.

```
1 | procedure TForm1.ButtonClick(Sender: TObject);
2 | begin
3 |   Edit3.Text := FloatToStr(Calculadora.Soma.Operacao(StrToCurr(Edit1.Text), StrToCurr(Edit2.Text)));
4 | end;
```

Ao olhar esse código isso me incomoda muito, e você não sabe o quanto isso incomoda. É casting de variáveis, é quando você tem que trocar o tipo de uma variável, de um retorno, como estamos fazendo no código acima.

Eu procuro deixar meus códigos o mais claro possível! Existem inúmeras formas de deixar seu código mais legível.

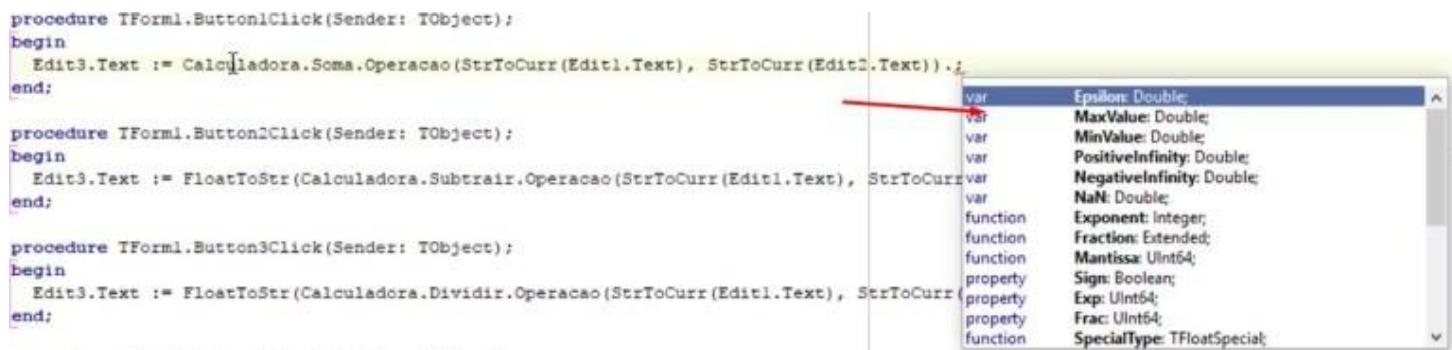
Vamos ver na prática de como trabalhar com class helper e como podemos usá-las para deixar nossos códigos mais legíveis.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS HELPER}

```
while not dmDados.SQL1.Eof do
begin
  Operador := TOperador.C...
```

No nosso código acima estamos convertendo de *float* para *string*, e de *string* para *currency*, na transformação de *float* para *string*, porque o *float* já possui algumas propriedades dentro dele.



A screenshot of the Delphi IDE showing a code completion dropdown for the *float* type. The dropdown lists various properties and methods, with a red arrow pointing to the *Var* section. The properties listed include *Epsilon*, *.MaxValue*, *.MinValue*, *PositiveInfinity*, *NegativeInfinity*, *NaN*, *Exponent*, *Fraction*, *Mantissa*, *Sign*, *Exp*, *Frac*, and *SpecialType*.

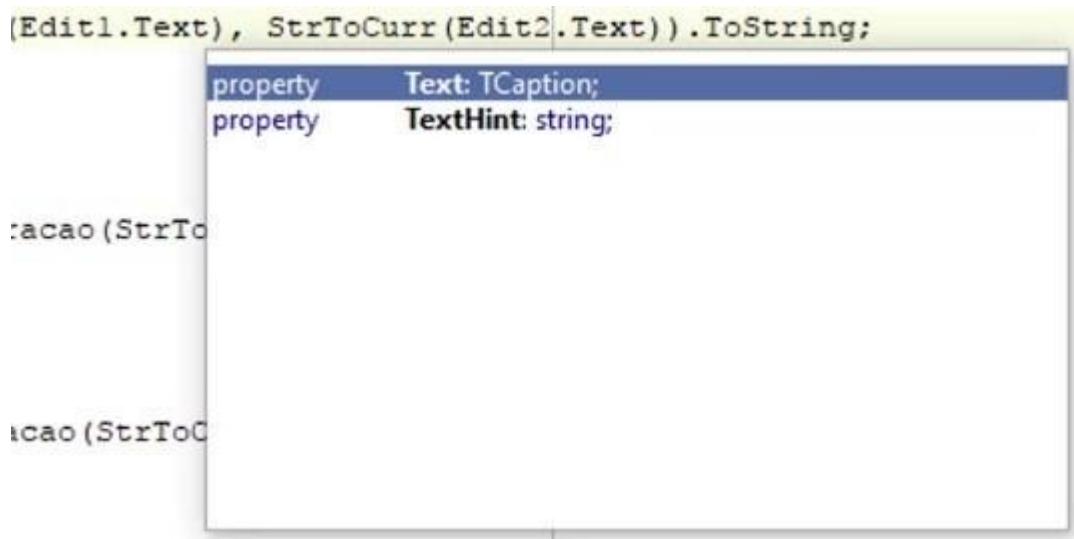
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit3.Text := Calculadora.Soma.Operacao(StrToCurr(Edit1.Text), StrToCurr(Edit2.Text));
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Edit3.Text := FloatToStr(Calculadora.Subtrair.Operacao(StrToCurr(Edit1.Text), StrToCurr(Edit2.Text)));
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  Edit3.Text := FloatToStr(Calculadora.Dividir.Operacao(StrToCurr(Edit1.Text), StrToCurr(Edit2.Text)));
end;
```

Um dos retornos do *float* é o *ToString*, ou seja, ele já faz a conversão sem precisar fazer um *casting* dentro do meu código.

Porém eu tenho no *edit* um *text* que é no tipo *TCaption*.



A screenshot of the Delphi IDE showing a code completion dropdown for the *TCaption* type. The dropdown lists properties *Text* and *TextHint*. The *Text* property is highlighted with a blue selection bar.

```
(Edit1.Text), StrToCurr(Edit2.Text)).ToString;
```

property	<b>Text: TCaption;</b>
property	TextHint: string;

```
:acao(StrToC
:acao(StrToC
```

Ele não possui nada agregado, então teríamos que fazer esse *casting* na mão, só que é aí que entra as classes helpers, já pensou em criar o seu próprio *TString* para o *TCaption*? Você pode criar agora.

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS HELPER}

```
while not dmDados.QueryUsuarios.EOF do begin
  Operador := TOperador(dmDados.QueryUsuarios);
  Operador.Nome := dmDados.QueryUsuarios.FieldByName('NOME').AsString;
```

Na classe calculadora, iremos criar uma classe helper.

```
1  TCaptionHelper = record helper for TCaption
2    function ToCurrency : Currency;
3  end;
4
5  ...
6
7  function TCaptionHelper.ToCurrency : Currency;
8  begin
9    Result := StrToCurr(Self);
10   end;
```

No código acima estamos criando um *helper* para classe *TCaption*, e dentro de nossa classe criamos os métodos que quisermos, só não podemos criar propriedades, não podemos criar novos atributos a classe, eu posso criar métodos para trabalhar com atributos já existentes.

Observe no código que estamos convertendo um texto, que é o *Self*, para o tipo *Currency*, e o nosso novo método retorna um *Currency*.

Agora no nosso primeiro código nós iremos fazer uns pequenos ajustes.

Com essa implementação o próprio compilador já identifica essa implementação para nós.

The screenshot shows a Delphi IDE interface. A tooltip is displayed above the code editor, containing the text: 'Edit1.Text, StrToCurr(Edit2.Text).ToString;'. Below the code editor, the Delphi code is visible, showing a class helper definition for *TCaption* with a *ToCurrency* method. The method signature is highlighted in blue: 'function ToCurrency: Currency;'. This visual cue from the IDE indicates that the compiler has recognized the helper implementation.

```
1  TCaptionHelper = record helper for TCaption
2    function ToCurrency : Currency;
3  end;
4
5  ...
6
7  function TCaptionHelper.ToCurrency : Currency;
8  begin
9    Result := StrToCurr(Self);
10   end;
```

# RECURSOS AVANÇADOS NO DELPHI

## {CLASS HELPER}

Agora só alterarmos nosso código e veja como ele ficou.

```
1 | procedure TForm1.ButtonClick(Sender: TObject);
2 | begin
3 |   Edit3.Text := Calculadora.Soma.Operacao(Edit1.Text.ToDouble, Edit2.Text.ToDouble).ToString;
4 | end;
```

Agora temos toda nossa operação de forma contínua, não preciso ficar fazendo casting dentro da minha camada de visão, isso não é legal, você tem que entregar para quem for trabalhar com seu formulário, as suas interfaces, seus métodos já funcionando perfeitamente, sem que tenha que ser alterado nada, ou ficar fazendo casting, as técnicas de orientação a objeto estão aí para lhe auxiliarem.

**RECURSOS  
AVANÇADOS  
NO DELPHI**



**{CENTRALIZANDO  
AS EXCEPTIONS}**

# RECURSOS AVANÇADOS NO DELPHI

## {CENTRALIZANDO AS EXCEPTIONS}

Neste e-book, eu falei de como usar forms VCL e FMX no mesmo projeto.

Neste capítulo, vou te passar mais uma dica. Irei tratar um problema recorrente que muitos de nós programadores Delphi enfrentamos, que é o tratamento de exceções ou tratamentos de erros dentro do software.

### Como você faz hoje o seu tratamento de erro?

Pode ser que você tenha um monte de métodos espalhados no seu software para gravar um *log*, em cada parte do seu software tem um *try exception* para gravar esses *logs*.

Estou aqui para lhe dizer que **não tem necessidade nenhuma de fazer isso**.

Para que possamos chegar a essa conclusão, precisamos remeter o que eu sempre prego na comunidade Delphi.

Vá a fundo e entenda o que você faz dentro do código. Não se limite a montagem de formulários, entenda o que está acontecendo, não se conforme, seja um eterno inconformado, essa dor da inconformidade é o que irá fazer você crescer.

Para chegarmos a solução para gravação de logs genéricas, você precisa saber o que está acontecendo, o que está acontecendo por trás das rotinas do Delphi, tudo como funciona, o que é um evento, o que é um ponteiro, etc...

# RECURSOS AVANÇADOS NO DELPHI

## {CENTRALIZANDO AS EXCEPTIONS}

Graças a tudo isso, nós conseguimos de forma única tratar todas as exceções dentro de uma única classe dentro do nosso projeto.

Primeiro, vamos criar um novo projeto VCL.

Com esse nosso novo projeto criado, iremos criar uma nova *Unit* e iremos salvá-la como *Exception*.

```
1 type
2   TException = class
3     private
4       { private declarations }
5     public
6       constructor Create;
7       procedure TrataException(Sender : TObject; E : Exception);
8     end;
```

Ao observar o código, você deve estar se perguntando, de onde você tirou isso dai e para que você quer isso?

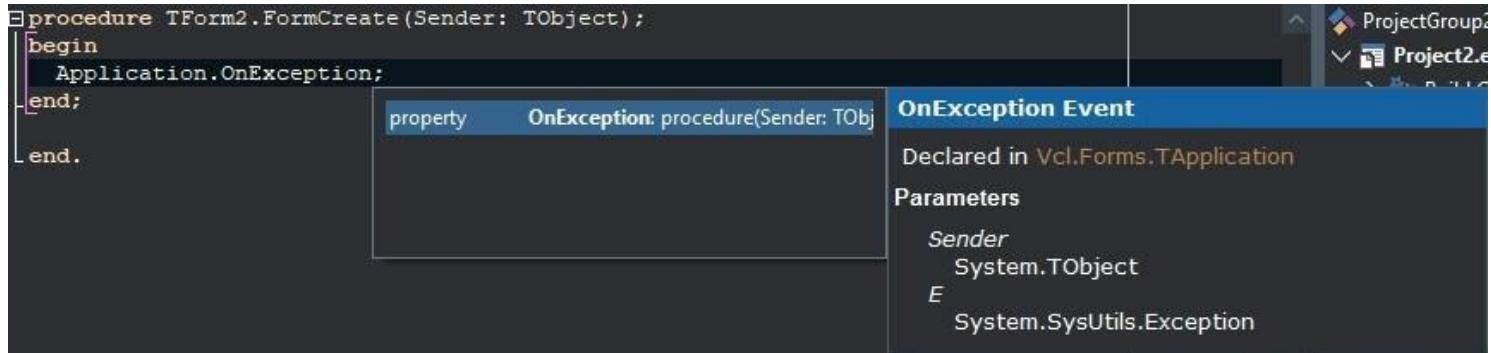
Já já você irá entender melhor...rsrsr

O comando *Application.OnException* diz que quando acontecer alguma exceção na minha aplicação irá executar um evento.

Se você reparar na imagem à seguir, a descrição de método é um evento dentro do Delphi, esse *Exception* está esperando receber uma procedure que tenha dois parâmetros, um *TObject*, e um *Exception*, para que ele possa iniciar uma referência para memória, assim todas as vezes que acontecer um *Exception* ele irá nesse endereço de memória e executa a procedure que está lá.

# RECURSOS AVANÇADOS NO DELPHI

## {CENTRALIZANDO AS EXCEPTIONS}



E é isso que vamos fazer, iremos criar uma procedure e vamos dizer para o `Application.onException` que toda vez que acontecer uma exceção, irá chamar o método que está aguardando tal requisição.

Iremos fazer alguma alterações na classe `Exception` que acabamos de criar.

```
1 type
2   TException = class
3     private
4       FLogFile : String;
5     public
6       ...
7       procedure GravarLog(value : String);
8     ...
9     constructor TException.Create;
10    begin
11      FLogFile := ChangeFileExt(ParamStr(0), '.log');
12      Application.onException := TrataException;
13    end;
14
15   procedure TException.GravarLog(value: String);
16   var
17     txtLog : TextFile;
18   begin
19     AssignFile(txtLog, FLogFile);
20     if FileExists(FLogFile) then
21       Append(txtLog)
22     else
23       Rewrite(txtLog);
24     Writeln(txtLog, FormatDateTime('dd/mm/YY hh:mm:ss - ', now) + value);
25     CloseFile(txtLog);
26   end;
27
```

# RECURSOS AVANÇADOS NO DELPHI {CENTRALIZANDO AS EXCEPTIONS}

```
28 procedure TException.TrataException(Sender: TObject; E: Exception);
29 begin
30     GravarLog('=====');
31     if TComponent(Sender) is TForm then
32     begin
33         GravarLog('Form: ' + TForm(Sender).Name);
34         GravarLog('Caption: ' + TForm(Sender).Caption);
35         GravarLog('Error: ' + E.ClassName);
36         GravarLog('Error: ' + E.Message);
37     end
38     else
39     begin
40         GravarLog('Form: ' + TForm(TComponent(Sender).Owner).Name);
41         GravarLog('Caption: ' + TForm(TComponent(Sender).Owner).Caption);
42         GravarLog('Error: ' + E.ClassName);
43         GravarLog('Error: ' + E.Message);
44     end;
45     GravarLog('=====');
46     Showmessage(E.Message);
47 end;
48
49 var
50     MinhaException : TException;
51     initialization;
52     MinhaException := TException.Create;
53     finalization;
54     MinhaException.Free;
```

Vamos analisar o código acima.

No Create eu passo o método *ChangeFileExt* para a variável *FLogFile* que adicionamos em nossa classe.

Esse método *ChangeFileExt* está recebendo dois parâmetros, o *ParamStr(0)*, que sempre irá retornar o caminho completo da minha aplicação, e uma string '.log', esse método *ChangeFileExt* só irá trocar a extensão do arquivo para o qual definimos no segundo parâmetro.

Desta forma eu tenho o meu arquivo de log com o mesmo nome do meu executável.

# RECURSOS AVANÇADOS NO DELPHI

## {CENTRALIZANDO AS EXCEPTIONS}

Na segunda linha do meu *Create* temos aquele comando que comentei no início do capítulo, o *Application.onException*, que por sua vez recebe o meu método *TrataException*.

Logo no método *TratarException*, poderíamos criar diversas formas para o tratamento das exceções, como por exemplo, enviar por e-mail, gravar um arquivo, nós podemos especializar isso de diversas formas.

Mas de exemplo só criei uma rotina de criação de arquivo simples, onde criamos um método, o *GravarLog*, que irá receber um valor string e irá trabalhar a criação do arquivo texto.

Dentro do nosso *TratarException*, simplesmente verificar se o objeto que chegou é um *TForm* eu pego e gravo os valores dentro do meu arquivo texto com as informações que preciso para tal situação.

E caso não seja um *TForm* fazemos um tratamento bem simples, onde estamos forçando para que ele pegue as propriedades do componente que deu o erro.

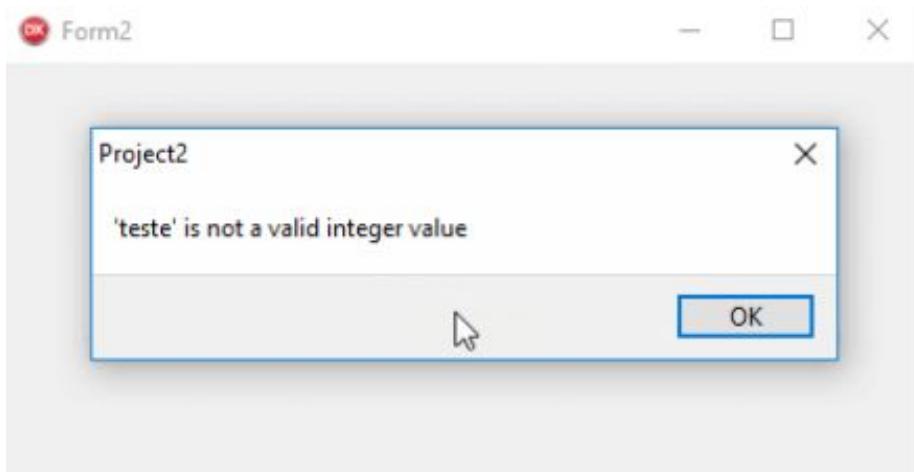
No final do método coloquei um *Showmessage* para que ao executarmos o nosso aplicativo possamos ver a mensagem de erro.

Vamos colocar um botão no formulário e iremos forçar um erro para que tenhamos nosso teste ok.

```
1 procedure TForm2.Button1Click(Sender: TObject);
2 var
3     I : integer;
4 begin
5     i := strtoint('teste');
6 end;
```

# RECURSOS AVANÇADOS NO DELPHI {CENTRALIZANDO AS EXCEPTIONS}

Veja só após executarmos nosso projeto.



Viu a mensagem de erro que saltou na tela?

Essa mensagem tem que está em nosso arquivo de log, vamos verificar se ele foi criado e seu conteúdo.

Outros (D:) > BlogFontes > CentralizandoExcecoes > Win32 > Debug			
Nome	Data de modificaç...	Tipo	Tamanho
Project2	09/08/2019 14:56	Aplicativo	12.147 KB
Project2	09/08/2019 14:57	Documento de Te...	1 KB
TrataException.dcu	09/08/2019 14:54	Arquivo DCU	5 KB
Unit2.dcu	09/08/2019 14:54	Arquivo DCU	6 KB

O arquivo foi criado perfeitamente com o nome do nosso executável.

```
09/08/19 14:57:26 - =====
09/08/19 14:57:26 - Form: Form2
09/08/19 14:57:26 - Caption: Form2
09/08/19 14:57:26 - Error: EConvertError
09/08/19 14:57:26 - Error: 'teste' is not a valid integer value
09/08/19 14:57:26 - =====
```

As informações que colocamos para serem salvas no arquivo de log estão aí, e olha a mensagem de erro que apareceu na tela.

Todas as vezes que ocorrer algum erro em nosso sistema irá gravar nesse log agora.

# RECURSOS AVANÇADOS NO DELPHI



{REMOVENDO **UNITS**  
DESNECESSÁRIAS}

# RECURSOS AVANÇADOS NO DELPHI

## {REMOVENDO UNITS DESNECESSÁRIAS}

Você como programador com toda certeza já deparou com o código-fonte do sistema que você desenvolveu e pensou:

Foi eu mesmo

Isso é natural. Enquanto você vai adquirindo mais conhecimentos e experiência em programação, conhecendo boas práticas, três camadas, e novas tecnologias é normal você querer programar seguindo esses padrões, e agora o seu software antigo será abandonado?

No entanto, este processo não é tão simples assim. O problema está no tempo para “refatorar” todo o sistema. Nem toda empresa de software dispõe de tempo para essa atividade, principalmente quando é um software de grande porte que apresenta demanda constante de manutenção. Este processo não é tão simples assim pois é necessário realizar testes e verificar se a alteração do código-fonte não irá gerar impactos em outros módulos do sistema. E há quem diga: Isso acontece muito!

Por exemplo, suponha que você trabalhou na refatoração no módulo de estoque de produtos, movendo rotinas duplicadas para funções parametrizadas e criando classes para agrupar atributos em comum. Ao término dessa refatoração, você reduziu as linhas de códigos em 40%. Ótimo, não é?

# RECURSOS AVANÇADOS NO DELPHI

## {REMOVENDO UNITS DESNECESSÁRIAS}

Porém, tenha em mente que outras unidades do seu sistema utilizam este módulo, como a entrada e saída de produtos. Se uma dessas unidades utiliza uma variável, método ou campo que foi refatorado, provavelmente a compilação apresentará erros. É claro, as ferramentas de desenvolvimento geralmente apontam erros de sintaxe e de referência caso o objeto não seja encontrado. Já a semântica, entretanto, não é validada por essas ferramentas, visto que se trata mais de uma questão voltada para domínio da aplicação. Neste caso, é imprescindível a execução de roteiros de testes e/ou testes automatizados das unidades que tenham alguma relação com o módulo alterado.

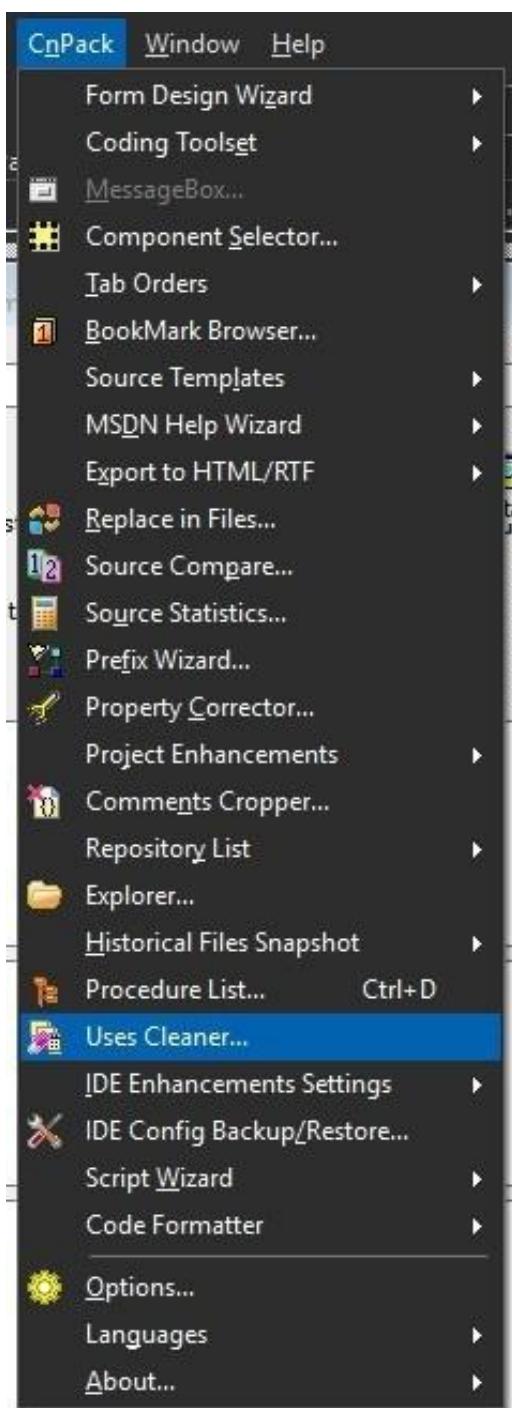
Neste capítulo, vou te mostrar uma ferramenta que pode te ajudar nesse processo, para você dar início a esse processo de refatoração.

# RECURSOS AVANÇADOS NO DELPHI

## {REMOVENDO UNITS DESNECESSÁRIAS}

Ferramenta do CnPack para remover Units não utilizadas

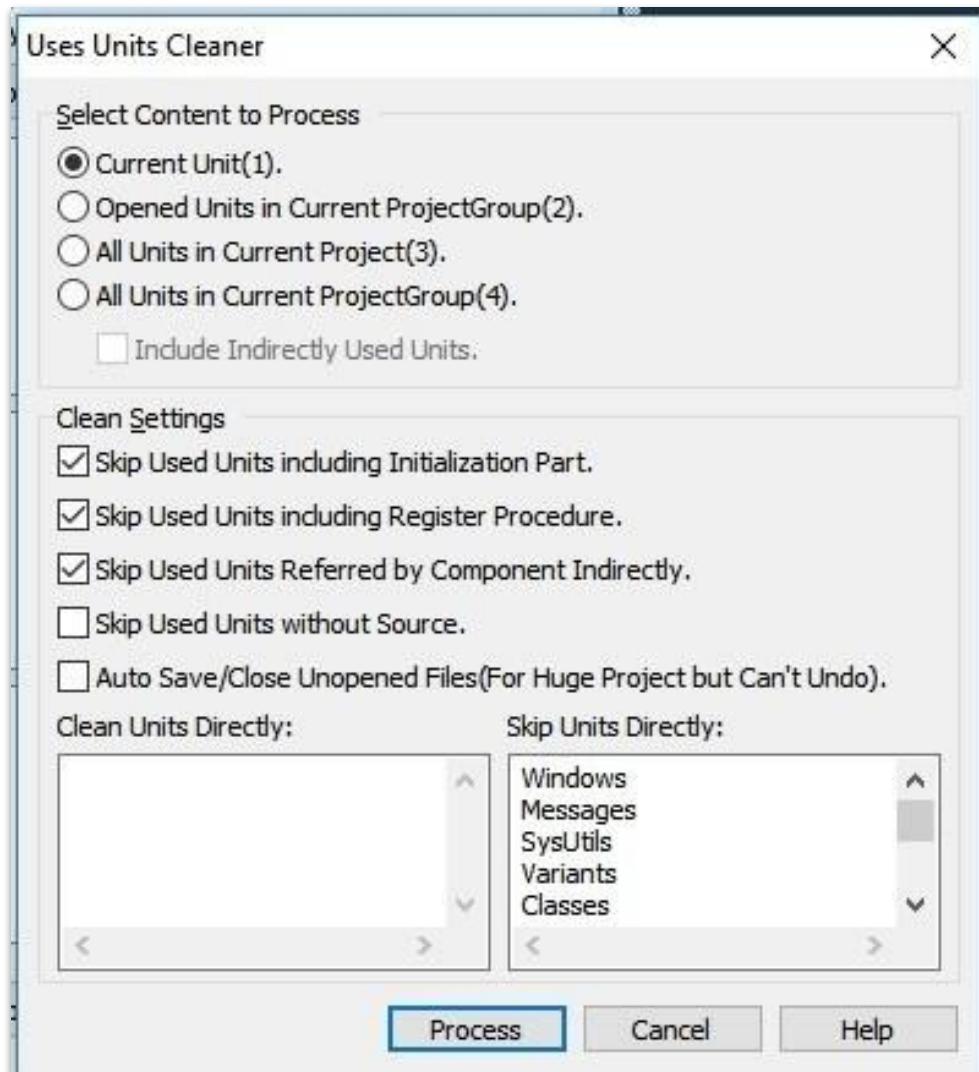
Após o CnPack instalado basta ir no seu menu, com seu projeto aberto, e clicar em *Uses Cleaner...*



# RECURSOS AVANÇADOS NO DELPHI

## {REMOVENDO UNITS DESNECESSÁRIAS}

Após isso ele irá abrir uma janela onde você terá as informações de units não utilizadas no seu projeto.



Após o CnPack processar as limpezas de units não usadas no seu projeto, ele irá compilar o mesmo, com isso você reduzirá bastante os lixos que deixamos no projeto.

# RECURSOS AVANÇADOS NO DELPHI



{QUÃO TÓXICO É  
O SEU CÓDIGO?}

# RECURSOS AVANÇADOS NO DELPHI

## {REMOVENDO UNITS DESNECESSÁRIAS}

Se você é alguém que escreve código, provavelmente conhece aquele momento em que olha para algum código que não escreveu, ou algum código que escreveu há muito tempo, e pensa “isso não parece bom”. Não é nem mesmo se o código faz o que deve fazer – demora um pouco mais para descobrir – ou se o código é muito lento. Mesmo que seja perfeitamente livre de erros e tenha um bom desempenho, existe algo na maneira como é escrito. Isso faz parte da qualidade interna de um sistema de software, algo que os usuários e gerentes de desenvolvimento não podem observar diretamente; no entanto, isso ainda os afeta, pois é difícil manter e ampliar códigos com baixa qualidade interna.

Agora, como desenvolvedor, você ajuda gerentes, executivos, ou até mesmo você que ocupa essa cadeira em sua empresa, quer entender a qualidade interna do código? Nós geralmente queremos um pouco mais do que “é horrível” antes de priorizar na limpeza do código sobre a implementação de novos recursos que geram valor do negócios diretamente. Ou até mesmo: como você descobre o quanto um código realmente é ruim em relação a algum outro código? Essas eram perguntas que eu mesmos me perguntava há alguns anos.

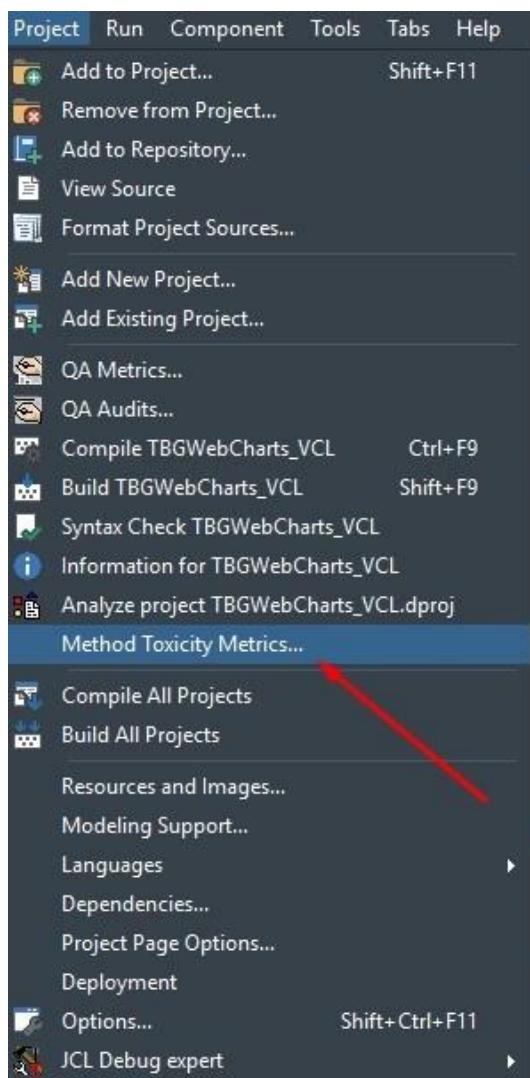
# RECURSOS AVANÇADOS NO DELPHI

## {REMOVENDO UNITS DESNECESSÁRIAS}

A resposta veio em algo que já tem dentro do **RAD Studio**, que pode nos auxiliar, e como eu priorizo muito na qualidade de código encontrei soluções que me auxiliam na hora de ter que refatorar um código.

Uma dessas é o **Method Toxicity Metrics** que ajuda na refatoração de métodos para que possamos aplicar padrões usados na programação funcional.

E para acessar é simples, basta ir no menu do **Rad Studio, Project > Method Toxicity Metrics**.



# RECURSOS AVANÇADOS NO DELPHI

## {REMOVENDO UNITS DESNECESSÁRIAS}

A tela que irá abrir mostra uma lista de todos os métodos junto com as principais métricas sobre cada método.

Method Name	Length	Parameters	If Depth	Cyclomatic Complexity	Toxicity
TForm1.Button7Click	57	1	1	13	1,754
TForm1.Button6Click	49	1	0	12	1,613
TForm1.Button4Click	48	1	0	12	1,600
TForm1.Button8Click	36	1	0	5	1,200
GravaFoto	24	3	2	4	1,050
ExibeFoto	11	3	1	2	0,396
JsonToDataset	11	2	1	2	0,354
TForm1.Button5Click	8	1	1	3	0,317
ExcluiFoto	2	2	1	2	0,242
Base64ToBlob	12	1	0	1	0,233
TForm1.Button1Click	10	1	0	1	0,208
BlobToBase64	9	1	0	1	0,196
TForm1.Button2Click	9	1	0	1	0,196
TForm1.Button3Click	9	1	0	1	0,196
TForm1.btnBase64Click	6	1	0	1	0,158
BlobBase64	6	1	0	1	0,158
base64Decode	4	1	0	1	0,133
base64Encode	4	1	0	1	0,133
Encode	2	1	0	1	0,108

- **Length:** O número de linhas no método ou, mais precisamente, o número de instruções de código no método.
- **Parameters:** o número de parâmetros na declaração do método.
- **If Depth:** A profundidade máxima de instruções if aninhadas presentes no método.

# RECURSOS AVANÇADOS NO DELPHI

## {REMOVENDO UNITS DESNECESSÁRIAS}

- **Cyclomatic Complexity:** O número mínimo de caminhos de código independentes através do método. Esse é o número mínimo de testes que devem ser executados para garantir que todas as linhas de código no método sejam testadas. Observe que a cyclomatic complexity não é o número total de caminhos de código independentes, mas o número mínimo de caminhos independentes.
- **Toxicity:** A toxicidade combina todas as métricas acima para criar um valor único de quão tóxico é um método.

A página **Method Toxicity Metrics** usa um plano de fundo vermelho para valores maiores que o limite definido. Você pode personalizar o limite para cada métrica na página de opções **Toxicity Metrics**. O limite do valor de Toxicidade é sempre 1, você não pode alterá-lo.

Você pode clicar duas vezes em qualquer método na lista para abrir sua implementação.

Seguindo essas métricas e usando o que há de melhor no Rad Studio, você pode alcançar um grau de maturidade em seus códigos, trazer o que tem de melhor na programação funcional, saindo de um código legado e totalmente acoplado para um código mais coeso e de fácil manutenção.

# CONHEÇA O CLUBE DE PROGRAMADORES DELPHI

## O que é o Clube?

Um clube de programadores com amplo conhecimento, tutelados por um dos MVP's de maior destaque no Brasil, compartilhando conhecimento por um valor super justo e acessível.



O Clube de Programadores Delphi tem a missão de fazer parte da sua vida profissional, lhe ajudando e capacitando nos mais diversos temas no mundo Delphi, para levar a sua carreira para um outro patamar.

[CLIQUE E CONHEÇA O CLUBE](#)

## O que você encontrará no Clube?

Escolhemos a dedo 8 competências que consideramos essenciais para todo desenvolvedor Delphi



Preparatório para  
Certificação Delphi



Componentes



Inovação



Recursos da  
Linguagem



Ferramentas de  
Terceiros



Firemonkey



Automação  
Comercial



Conteúdos  
Exclusivo



Thulio  
Bittencourt