

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi .NET



## Generics no RAD Studio 2007

Entendendo e utilizando Generics em suas aplicações .NET



**Manoel Edesio B da Silva**

([manoel.edesio@hkconsultoria.com.br](mailto:manoel.edesio@hkconsultoria.com.br))

detém as certificações em Borland C++ Builder, Borland Delphi for Win32 e Borland Delphi for .NET. Palestrou para a Borland em Delphi Meetings em alguns estados brasileiros, palestrante na Borland Conference de 2004, 2005, 2006 e 2007. Atualmente é diretor da HK Consultoria, empresa especializada em consultoria, treinamentos customizados e desenvolvimento de software em produtos Borland. Além dos quesitos anteriores a HK atua com projetos especializados em mobilidade com utilização de celulares e PDA'S.

**G**enerics é um termo utilizado para se definir estruturas com tipos genéricos, para representar classes, métodos de classe etc; podendo os mesmos serem parametrizados com qualquer tipo de dado. O conceito do Generics já existe em linguagens como Java e C++. Em C++ o recurso é conhecido como *templates*. Todos nós já tivemos sérios problemas em escrita de código de forma genérica; principalmente todos os amantes de POO. Quando procuramos escrever um código portátil às vezes nos deparamos com situações onde precisamos representar certo tipo de dado, porém esse tipo precisa ser modificado em *runtime* de acordo com a situação.

Algumas pessoas podem até achar que *generics* é algo parecido com *instanciamento* de classes descendentes em tipos ancestrais ou algo similar feito com *interfaces*, mas você poderá conferir em nosso artigo que esse maravilhoso recurso vai muito além do que o pro-

gramador Delphi está acostumado. Ele irá simplesmente transformar a vida de todos os programadores em Delphi.

Esse recurso atualmente está disponível apenas para plataforma .NET. Já respondendo a pergunta de quando estará disponível para Win32, ainda não temos nada definido pela CodeGear. Mas já sabemos que em breve o recurso estará disponível para a nossa querida plataforma Win32.

### Mais como assim? Qualquer tipo de dado?

Imagine que a partir de agora os tipos de dados de seus parâmetros, por exemplo, podem ser definidos em *runtime*, acredita? É verdade caros amigos *delphianos*, e é exatamente isso que nos dará a flexibilidade para fazer coisas absurdas em nossa linguagem.

Para ficar mais claro vamos criar uma nova aplicação utilizando o menu `File\New>VCL Forms Application - Del-`

*phi* for .NET no CodeGear RAD Studio 2007 e desenhar a interface da **Figura 1**. Salve o formulário como “uPrincipal.pas” e modifique seu *Name* para “frmPrincipal”. O projeto salve como “Generics.dpr”. Em seguida vamos adicionar uma nova *Unit* utilizando o menu *File\New>Other>Delphi for .NET Projects>New Files>Unit*, para definir a estrutura presente na **Listagem 1** e salve-a como “untPessoa.pas”.

Agora imaginem o seguinte exemplo: iremos definir um *TList* para manipular objetos. Essa classe dá suporte a qualquer tipo de objeto em uma mesma lista, porém temos muitos problemas de desempenho através de conversões implícitas. Para recuperar as informações da lista precisamos sempre utilizar o recurso de *typecast* para manipular os objetos, além disso, se definirmos uma lista, por exemplo, para manipular objetos de uma classe chamada *TPessoa* não conseguimos garantir que serão inseridos apenas objetos derivados dessa classe. Com isso podemos inserir qualquer objeto em uma lista do tipo *TList* e como não temos restrição do tipo adicionado se fizermos alguma besteira, só vamos saber em *runtime* no momento de manipular o objeto através do *typecast*.

Para testar o exemplo vamos acessar o nosso formulário e dar um duplo clique no botão *criar* e adicionar o código da **Listagem 2**. Digitado o código vamos executar a aplicação pressionando *Shift+Ctrl+F9* e clicar no botão *criar* para testarmos o exemplo. Como estamos adicionando um *TButton* na lista e fazendo um *typecast* para *TPessoa* em

#### Listagem 1. Definição da classe TPessoa

```
unit untPessoa;
interface
type
  TPessoa = class
  private
    FName: string;
  public
    constructor Create(PNome:string);
    procedure setNome(const Value: string);
    property Nome:string read FName write setNome;
  end;
implementation
constructor TPessoa.Create(PNome: string);
begin
  inherited Create;
  Nome := PNome;
end;
procedure TPessoa.setNome(const Value: string);
begin
  FName := Value;
end;
end.
```

#### Listagem 2. Manipulação através de um TList (Click do botão criar)

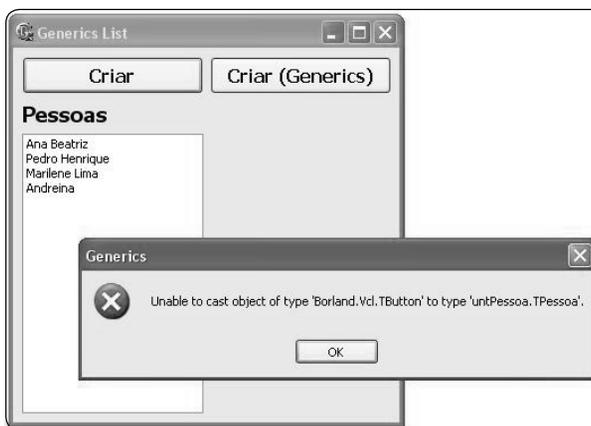
```
procedure TfrmPrincipal.Button1Click(Sender: TObject);
var
  PersonList : TList;
  o : TObject;
  p : TPessoa;
begin
  PersonList := TList.Create;
  PersonList.Add(TPessoa.Create('Ana Beatriz'));
  PersonList.Add(TPessoa.Create('Pedro Henrique'));
  PersonList.Add(TPessoa.Create('Marilene Lima'));
  PersonList.Add(TPessoa.Create('Andreina'));
  { A linha a baixo provocará um erro em Runtime }
  PersonList.Add(button2);
  for o in PersonList do
  begin
    p := o as TPessoa;
    lstbxPessoas.Items.Add(p.Nome);
  end;
end;
```

#### Listagem 3. Manipulação de uma lista com Generics (Click do botão criar Generics)

```
procedure TfrmPrincipal.Button2Click(Sender: TObject);
var
  PersonList : List<TPessoa>;
  P : TPessoa;
begin
  PersonList := List<TPessoa>.Create;
  PersonList.Add(TPessoa.Create('Ana Beatriz'));
  PersonList.Add(TPessoa.Create('Pedro Henrique'));
  PersonList.Add(TPessoa.Create('Marilene Lima'));
  PersonList.Add(TPessoa.Create('Andreina'));
  { Erro em Tempo de Compilação visto que o tipo foi definido como TPessoa }
  PersonList.Add(button2);
  for p in PersonList do
  begin
    lstbxPessoas.Items.Add(p.Nome);
  end;
end;
```



**Figura 1.** Exemplo de tela para teste da classe TList



**Figura 2.** Exceção ocorrida em runtime ao fazer typecast do TButton para TPessoa em um TList



*runtime* teremos uma exceção de acordo com a **Figura 2**.

## Primeiros passos

Considerando o exemplo anterior vamos implementar a mesma idéia através de *Generics*. De um duplo clique no botão *Criar(Generics)* e adicione o código da **Listagem 3**. Repare que no início do código temos o item *var* onde a definição do tipo genérico é feita entre "<" e ">". Como o tipo *List* dá suporte nativo a *Generics* só precisamos informar qual é o tipo que ele irá suportar. Em nosso código estamos dizendo com a definição *List <TPessoa>* que a lista é composta apenas por objetos do tipo *TPessoa*. A partir do momento da definição, a lista só irá aceitar objetos desse tipo; inclusive podemos observar isso no *Code Completion* do RAD Studio na **Figura 3**.

Ao tentar inserir um *TButton* ou qualquer outro objeto que não seja do

tipo passado no *Generic Type*, o compilador irá recusar o mesmo em tempo de compilação, não sendo necessário executar a aplicação para receber o erro em *runtime*.

Além disso, podemos reparar que ao recuperar o objeto da lista não foi necessário fazer uso de *typecast*. Como o compilador reconhece o tipo automaticamente ele já tem todo o conteúdo de sua infra-estrutura, além de tudo temos o nosso querido *Code Completion* ao manipular os objetos da lista. É realmente incrível como o compilador realiza essa tarefa.

## Um exemplo cotidiano

Um exemplo muito comum para demonstrar ainda o melhor o uso de *Generics* são as definições de *Pair Types*. Em nosso dia-a-dia sempre precisamos utilizar *pares combinados* de valores para manipulação através de classes. Porém em uma hierarquia de classes podemos

ter muitas combinações em tipos de valores; imagine, por exemplo, uma combinação de dois valores. Podemos ter dois valores como *string*, um *string* e um inteiro, um inteiro e um real, um como *TEndereco* e outro como *TPessoa* etc.

O tipo utilizado sempre vai depender do momento da utilização em nossas aplicações. Já pensou se precisássemos ter em cada classe pares de valores? Certamente programando da forma tradicional teríamos que implementar essa particularidade em cada aplicação nossa, em cada classe etc.

Então qual seria a melhor maneira de resolver esse problema? A resposta é: criar um tipo genérico que aceite dois tipos diferentes em *runtime*.

Para demonstrar esse exemplo crie uma nova aplicação *VCL forms Application – Delphi for .NET* com no exemplo anterior e desenhe uma tela como na **Figura 4**. E em seguida adicione uma nova *Unit* e salve a mesma com o nome de "unitTypeGeneric.pas". Nessa *unit* vamos adicionar uma classe chamada *TPair* com a definição da **Listagem 4**.

Perceba que a classe é definida e como parâmetro ao tipo temos dois *generic types* "TPair<TKey,TValue>" que classificamos como *TKey* e *TValue*, ao fazer isso estamos oferecendo a possibilidade de que no momento da utilização do tipo *TPair* poderemos informar quais os tipos serão utilizados no lugar do *TKey* e no lugar do *TValue*. Em continuidade ao código criamos duas propriedades uma chamada *Key* e uma chamada *Value*. Veja que as propriedades são dos tipos genéricos definidos na classe. Com isso ao se definir um tipo genérico, se quisermos manipular o mesmo devemos sempre referenciá-los por meio dos nomes.

```
PersonList.Add(TPessoa.Create('Ana Beatriz'));
PersonList.Add(TPessoa.Create('Pedro Henrique'));
PersonList.Add(TPessoa.Create('Marilene Lima'));
PersonList.Add(TPessoa.Create('Andreina'));
// PersonList.AddItem: TPessoa [?]; //Erro em Tempo de Compilação
```

**Figura 3.** Code Completion já reconhece o tipo definido

### Listagem 4. Definição da classe TPair

```
unit unitTypeGeneric;

interface
type
TPair<TKey, TValue> = class(TObject)
private
FKey: TKey;
FValue: TValue;
function GetKey: TKey;
function GetValue: TValue;
procedure SetKey(const Value: TKey);
procedure SetValue(const Value: TValue);
published
property Key: TKey read GetKey write SetKey;
property Value: TValue read GetValue write SetValue;
end;

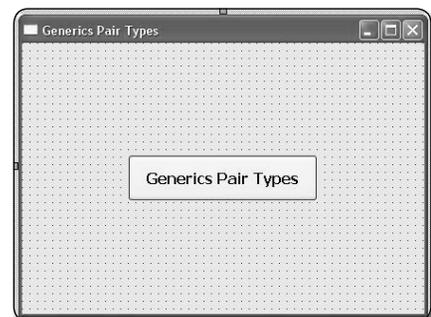
implementation

function TPair<TKey, TValue>.GetKey: TKey;
begin
result := FKey;
end;

function TPair<TKey, TValue>.GetValue: TValue;
begin
result := FValue;
end;

procedure TPair<TKey, TValue>.SetKey(const Value: TKey);
begin
FKey := Value;
end;

procedure TPair<TKey, TValue>.SetValue(const Value: TValue);
begin
FValue := Value;
end;
```



**Figura 4.** Interface da aplicação de exemplo Pair Types

Para utilizarmos agora a classe criada, vamos acessar a *unit* do nosso formulário principal e na seção *Type* antes da definição do *TForm* vamos adicionar a definição conforme a seguir:

```
type
  TSSPairOne = TPair<string,string>;
  TSSPairTwo = TPair<integer,real>;
```

Além disso, não podemos esquecer de adicionar na seção *interface* da *Unit* do formulário a referência para a *Unit* *unitTypeGeneric* na cláusula *Uses*. Esse código está definindo dois tipos, um deles contém um par de chaves onde o *Key* e *Value* são do tipo *string*, já o segundo define o *Key* como *Integer* e o *Value* como *Real*. Agora para executarmos um teste utilizando os nossos tipos dê um duplo clique no botão *Generics* e adicione o código da **Listagem 5**.

Nesse código temos a definição de duas variáveis, uma que utiliza o tipo *TSSPairOne* e outra que faz uso do *TSSPairTwo*, na seqüência vamos utilizar as variáveis definidas. Instanciamos cada uma delas com a classe equivalente passando como parâmetro o tipo definido e em seguida preenchemos as propriedades com valores que obrigatoriamente tem que seguir a definição de cada tipo. Observe na **Figura 5** que o tipo de cada propriedade já é identificado pelo RAD Studio no *Code Completion*, e se for passado um tipo diferente da definição o próprio compilador já avisa ao programador gerando um erro no código.

## Trabalhando com Constraints

Já aprendemos como se definir um *Generic Type* e utilizar o mesmo informando o tipo suportado em *runtime*, porém da forma vista até agora o parâmetro sempre suportará qualquer tipo. Existe um recurso do *Generics* denominado *Constraints*, que é uma forma de gerar uma obrigatoriedade quanto ao tipo passado para um *Generics Type*. Esse recurso poderá ser aplicado a todas as formas de parametrização que inclui: classes, records, interfaces e ainda métodos parametrizados.

Para exemplificar o recurso vamos construir um exemplo onde o parâmetro passado ao *Generic Type* terá a obrigatoriedade de ser especificamente de um



```
FSSPairTwo := TPair<System.int32, System.Double>.Create;
FSSPairTwo.Key := 1;
FSSPairTwo.
end;
end.
```

procedure SetValue[const Value: Double];  
property Key: Integer;  
property Value: Double;  
constructor Create;  
function ToString: string;  
function Equals(obj: TObject): Boolean;

**Figura 5.** Code Completion exibindo os tipos definidos para as propriedades *Key* e *Value*

### Listagem 5. Implementando o Pair Types (Click do botão criar Generics)

```
procedure TfrmPrincipal.Button1Click(Sender: TObject);
var
  FSSPairOne: TSSPairOne;
  FSSPairTwo: TSSPairTwo;
begin
  FSSPairOne := TPair<System.string, System.string>
    .Create;
  FSSPairOne.Key := 'Idade';
  FSSPairOne.Value := '10';
  FSSPairTwo := TPair<System.int32, System.Double>
    .Create;
  FSSPairTwo.Key := 1;
  FSSPairTwo.Value := 4.5;
end;
```

### Listagem 6. Implementação do código da unit unitDefineGenericType

```
unit unitDefineGenericType;

interface

uses Dialogs;

type
  ICommunicate = interface
    procedure Speak;
  end;
  TMammalCom < T: ICommunicate > = class(TObject)
  private
    FMammal: T;
  public
    procedure DoSpeak;
    property Mammal: T read FMammal write FMammal;
  end;

  TMammal = class(TObject, ICommunicate)
  private
    FName: string;
  public
    procedure Speak;
    property Name: string read FName write FName;
  end;

  TAndroid = class
  end;

implementation

procedure TMammalCom < T > .DoSpeak;
begin
  Self.Mammal.Speak;
end;

procedure TMammal.Speak;
begin
  ShowMessage('Mammal Falando!!!');
end;
end.
```

```

procedure TForm1.Button2Click(Sender: TObject);
var
  MyMammal: TMammalCom<TMammal>;
  MeuAndroid: TMammalCom<TAndroid>;
begin
end;

```

Messages

- [DCC Warning] untFrmGenerics01.pas(6): W1005 Unit 'Borland.Vcl.Windows' is specific to a platform
- [DCC Warning] untFrmGenerics01.pas(6): W1005 Unit 'Borland.Vcl.Messages' is specific to a platform
- [DCC Warning] untFrmGenerics01.pas(34): H2164 Variable 'TempNome' is declared but never used in 'TForm1.Button1Click'
- [DCC Error] untFrmGenerics01.pas(80): E2514 Type parameter 'T' must support interface 'IComunicate'
- [DCC Error] Generics01.dpr(11): F2063 Could not compile used unit 'untFrmGenerics01.pas'

Figura 6. Evento clique do botão Generics Constraints

**Listagem 7.** Implementação do código da unit `unitGenericTypeConstructor`

```

unit unitGenericTypeConstructor;

interface

uses Dialogs;

type

TNeedsConstructorGenericType < TypeConstructor: constructor > = class
private
  FDados: TypeConstructor;
public
  constructor Create;
  destructor Destroy; override;
  function GetData: TypeConstructor;
  procedure SetData(Value: TypeConstructor);
  property Dados: TypeConstructor read GetData write SetData;
end;

TRecordNoConstructor = record
  SomeInteger: integer;
end;

TClassHasConstructor = class
public
  SomeInteger: integer;
  constructor Create;
end;

implementation

constructor TNeedsConstructorGenericType < TypeConstructor > .Create;
begin
  inherited;
  Dados := TypeConstructor.Create;
end;

destructor TNeedsConstructorGenericType <TypeConstructor>.Destroy;
begin
  inherited;
end;

function TNeedsConstructorGenericType <TypeConstructor>.GetData: TypeConstructor;
begin
  result := FDados;
end;

procedure TNeedsConstructorGenericType <TypeConstructor>.SetData(Value:
TypeConstructor);
begin
  FDados := Value;
end;

constructor TClassHasConstructor.Create;
begin
  inherited;
end;

end.

```

tipo, e com isso vamos notar que ao manipular esse *Generic Type* teremos que obedecer essa regra, caso contrário não vamos conseguir utilizar o tipo criado.

Vamos lá mãos na massa agora! Crie uma nova aplicação *VCL forms Application – Delphi for .NET* utilizando o mesmo esquema dos exemplos anteriores. Apenas insira um `TButton` ao formulário principal e salve-o como “uPrincipal.pas”. Em seguida vamos criar uma nova *Unit* chamada “unitDefineGenericType.pas” com a estrutura da **Listagem 6**. Dentro da *Unit* definimos individualmente uma *interface* chamada *IComunicate* com um método definido chamado *Speak*, na seqüência temos a definição de uma classe chamada *TMammalCom* que define um tipo genérico. Mais percebam que temos uma definição por *Constraint* na classe *TMammalCom* <T: *IComunicate*>. Esse *Constraint* está informando que o tipo que será definido ao *Generic Type* obrigatoriamente terá que implementar a *interface IComunicate*. Em seguida definimos duas classes, uma que implementa a interface e outra que não dá suporte a mesma.

Após definição da *Unit* no formulário principal vamos dar um duplo clique no botão *Generic Constraints* e implementar o código a seguir.

```

procedure TForm1.Button1Click(Sender:
TObject);
var
  MyMammal: TMammalCom < TMammal > ;
  MeuAndroid: TMammalCom < TAndroid > ;
begin
end;

```

Você poderá reparar que na primeira linha definimos uma variável para o tipo genérico passando como parâmetro a classe *TMammal* que dá suporte a *interface* e na seqüência definimos uma segunda variável passando a classe *TAndroid* que não tem suporte a *interface*. Ao compilar podemos observar ainda na mesma figura o erro gerado pelo compilador informando que o tipo *TAndroid* não dá suporte à *interface* (**Figura 6**).

**Constraints com construtor**

Continuando o exemplo de *Constraints* vamos exemplificar o tipo parâmetro *Constructor* na definição de classes. Com esse tipo de *constraints* vamos exigir que

na passagem de parâmetro do tipo seja obrigatoriamente exigido um construtor público. Caso contrário o compilador irá nos informar que o parâmetro não é válido. Agora repita os passos anteriores e crie um novo projeto *VCL forms Application – Delphi for .NET* inserindo apenas um botão ao formulário principal. Crie uma nova *Unit* com o nome “unitGenericTypeConstructor.pas” e defina seu código como na **Listagem 7**.

Estamos definindo nessa *Unit* uma classe chamada *TNeedsConstructorGenericType* que define um *Generic Type* com um constraint *constructor*, em seguida temos a definição de uma classe e de um *record*. E a nossa classe tem um método construtor definido com visibilidade pública. Em seguida vamos até o formulário de nossa aplicação e dar um duplo clique no botão *Generic Type Constructor* e implementar o código da **Listagem 8**.

Estamos definindo duas variáveis baseados no tipo *TNeedsConstructorGenericType* porém um tipo passado como parâmetro é um *record* e o outro é uma classe, você poderá reparar que ao tentar compilar o projeto vamos receber do compilador do RAD Studio a mensagem de erro da **Figura 7** onde estamos sendo informados que o tipo parâmetro *TypeConstructor* precisa de um construtor público. Para executar a aplicação comente a linha que contém a definição *MyNoConstructorType:TNeeds*

```
30 procedure TForm1.Button1Click(Sender: TObject);
    var
        MyConstructorType:TNeedsConstructorGenericType<TClassHasConstructor>;
33     MyNoConstructorType:TNeedsConstructorGenericType<TRecordNoConstructor>;
    begin
        MyConstructorType:= TNeedsConstructorGenericType<TClassHasConstructor>.Create;
    end;
end;
```

Messages

```
[DCC Warning] unitFrmGenerics01.pas(6): W1005 Unit 'Borland.Vcl.Windows' is specific to a platform
[DCC Warning] unitFrmGenerics01.pas(6): W1005 Unit 'Borland.Vcl.Messages' is specific to a platform
[DCC Error] unitFrmGenerics01.pas(33): E2513 Type parameter 'TypeConstructor' must have public parameterless constructor
[DCC Error] Generics01.dpr(10): F2063 Could not compile used unit 'unitFrmGenerics01.pas'
```

**Figura 7.** Erro gerado pelo Generic Type Constructor

#### Listagem 8. Click do botão Generic Type Constructor

```
procedure TfrmPrincipal.Button1Click(Sender: TObject);
var
    MyConstructorType: TNeedsConstructorGenericType
    <TClassHasConstructor>;
    MyNoConstructorType: TNeedsConstructorGenericType
    <TRecordNoConstructor>;
begin
    MyConstructorType := TNeedsConstructorGenericType
    <TClassHasConstructor>.Create;
    try
        MyConstructorType.Dados.SomeInteger := 44;
        ShowMessage('O Valor é: ' + MyConstructorType
        .Dados.SomeInteger.ToString);
    finally
        MyConstructorType.Free;
    end;
end;
```

*ConstructorGenericType<TRecordNoConstructor>;* e rode a aplicação.

## Conclusão

Por meio desse artigo podemos concluir que cada dia que passa estão sendo incluídos novos recursos na linguagem para facilitar o nosso dia-a-dia. O *generics* é um recurso utilizado junto a programação orientado a objetos que consegue

reduzir drasticamente o esforço de programação quando temos a necessidade de manipular tipos que possam se alternar em determinados momentos. Espero ter contribuído através desse artigo para um melhor esclarecimento sobre esse interessante assunto e já incentivado a todos delphianos a inclusão do recurso em seus projetos .NET. ●

