C# - Generics- Artigo easy .net Magazine 11

O artigo aborda como utilizar Generics, com foco em coleções de dados. Será apresentada uma introdução ao recurso e alguns exemplos práticos, inclusive utilizando de recursos anteriores aos Generics para entender seus benefícios.

Atenção: esse artigo tem um vídeo complementar. Clique e assista!

De que se trata o artigo

O artigo aborda como utilizar Generics, com foco em coleções de dados. Será apresentada uma introdução ao recurso e alguns exemplos práticos, inclusive utilizando de recursos anteriores aos Generics para entender seus benefícios.

Para que serve

Generics é um poderoso recurso incluído na plataforma .net desde a versão 2.0 do framework, com ele podemos criar classes e métodos reutilizáveis com mais eficiência, seu uso mais comum estão na manipulação de coleções fortemente tipadas.

Em que situação o tema é útil

A plataforma .net é repleta de suporte a manipulação de coleções de dados, mas foi na versão 2.0 que uma mudança significativa foi realizada, a inclusão de generics possibilitou criar coleções de forma mais eficiente, reduzindo ou eliminando a necessidade de conversão de tipos.

Generics

Generics são hoje uns dos principais fundamentos da programação para a plataforma .NET, de forma que podemos encontrar a aplicação do recurso em várias situações no .NET Framework. Permitem flexibilizar a forma como dados são tratados, pois é definido um parâmetro para um tipo. O artigo apresentará os motivos pelos quais os Generics surgiram, tratando de operações de Box, Unbox e conversões. Nos exemplos práticos os Generics são demonstrados com coleções, como List e Dictionary. Ao final criaremos uma coleção customizada.

Durante o desenvolvimento de um projeto procuramos sempre reutilizar o máximo de códigos, procurando evitar a repetição de linhas e com isso temos um código mais enxuto e flexível para mudanças. Felizmente as linguagens de programação mais utilizadas estão repletas de recursos que nos possibilitam realizar esta tarefa. Conhecer bem o funcionamento da linguagem é um início para criar bons projetos. Apesar de atualmente existir uma série de aprendizados para que o programador possa utilizar das melhores práticas de codificação e sempre ter um código bem elaborado, existem alguns recursos antigos que ainda são considerados poderosos para auxiliar no desenvolvimento de projetos, Generics é um deles.

O .net Framework é uma poderosa tecnologia, sua evolução está em constante crescimento e sempre atualizada com as mais recentes necessidades de desenvolvimento de software, madura e com suporte a uma séria de tecnologias. Hoje proporciona o desenvolvimento para os mais diversos tipos de aplicativos. Atualmente o Visual Studio é um dos IDEs mais poderosos do mercado, e o Visual C# cada vez está melhor, sendo considerada uma das linguagens de programação mais relevantes da atualidade.

Atualmente o .net framework está na versão 4.0, porém foi na versão 2.0 que o assunto deste tutorial foi adicionado ao framework. Nesta versão do .net framework houve diversas melhorias importantes em relação à versão anterior, e uma das mais relevantes e poderosas foi a inclusão de Generics. Com ele podemos criar um tipo especial que recebe como parâmetro outro tipo.

Em C# temos os tipos de valor, chamados de Value Types e os tipos de referência, chamados de Reference Types. Resumidamente, os Value Types armazenam um valor e os Reference Types armazenam uma referência aos dados. Os tipos de valores derivam implicitamente do System.ValueType. Os tipos de referência guardam nele o endereço da memória onde o objeto está registrado, somente o endereço e não o objeto real. Já as variáveis de tipos de valor (Values Type) contêm o próprio objeto.

Generics possibilitam a eliminação de conversão de tipos em sua aplicação. Para entender melhor o que seria uma conversão de tipos, vamos pensar no conceito de Box e Unbox. O Box seria quando adicionamos um elemento em uma caixa e fechamos, o Unbox seria quando abrimos e retiramos o elemento da caixa. Enquanto o elemento está dentro da caixa fechada, outras pessoas não saberão o que tem dentro, porém quando alguém receber a caixa e abrir, pode não estar preparado para seu conteúdo. De forma similar funcionam as conversões de tipos. Eu posso jogar qualquer valor dentro de um objeto (caixa), mas na hora de usar esse valor, eu vou precisar estar preparado para recebê-lo, ou seja, se for um tipo Double que está dentro do objeto, quando abrir a caixa eu preciso convertê-lo para Double e depois inserir em uma variável desse tipo.

Um exemplo, criando uma variável do tipo Object, podemos inserir um tipo Double dentro do elemento Object. Para criar a variável Double usamos, por exemplo:

```
double tipo_double = 1.20;
```

Para inserir dentro da variável Object:

```
Object box = tipo_double;
```

Dessa forma colocamos um tipo dentro da caixa. Para retirar da caixa e inserir novamente em uma variável Double, realizamos uma conversão forçada utilizando o tipo do objeto entre parênteses, antes de inserir na variável:

```
double tipo_double = (double)box;
```

Vimos o conceito de Box e Unbox, porém existe um grave problema em sua utilização, o compilador não apresentará erros em sua execução, caso o programador tenha colocado algum tipo na caixa diferente do tipo da variável que está esperando o valor. Imagine o mesmo exemplo anterior, ou seja, a caixa contém um tipo Double, mas a variável criada para receber o conteúdo seja do tipo DateTime, exemplo:

```
DateTime tipo = (DateTime)box;
```

Neste caso, o erro somente vai aparecer em tempo de execução, pois o compilador imagina que o programador saiba o que está fazendo, pois está realizando uma conversão forçada.

Com Generics eliminamos a necessidade de conversão de tipos em uma coleção, por exemplo, podemos utilizar um tipo genérico, que pode receber um tipo qualquer, e criar uma coleção fortemente tipada, ou seja, definindo o tipo que a coleção vai armazenar e com isso elimina a necessidade de uma conversão.

Você pode utilizar Generics para criar suas próprias classes, métodos, interfaces e eventos genéricos, mas seu uso mais comum está nas coleções (Collections). Dentre os principais benefícios dos Generics está a reutilização de código, pois com esse recurso você terá uma grande flexibilidade para trabalhar com tipos diferentes.

Os Generics estão agrupados no namespace System.Collections.Generics. As coleções genéricas são fortemente tipadas, onde somente um tipo de dados pertence à coleção. Ao tentar inserir outro tipo de dados, o compilador é quem vai apresentar o erro, que não mais acontece em tempo de execução, auxiliando muito o programador a evitar que o usuário receba mensagens de erro por falha no desenvolvimento.

A manipulação de coleções é uma das necessidades mais comuns em qualquer linguagem de programação. Um dos mecanismos mais simples para esse tipo de tarefa é o array. No entanto, ele tem algumas limitações, talvez a mais importante seja que o tamanho do array é fixado quando ele é criado. Para suprir essas necessidades e outras, melhorando a manipulação de coleções, o C# trouxe diversos tipos de coleções agrupadas em um namespace chamado System.Collections:

• ArrayList – Similar a um array, mas possibilitando crescer a coleção dinamicamente conforme os elementos são modificados;

• Stack – Uma coleção que aplica o princípio LIFO (Last In - First Out), o Stack permite manipular uma coleção em forma de pilha, onde o último item a entrar na pilha é o primeiro item a sair dela;

• Queue - Uma coleção que aplica o princípio FIFO (First In - First Out), o Queue permite manipular uma coleção em forma de fila, onde o primeiro item a entrar é o primeiro item a sair dela;

• HashTable – Uma coleção que armazena os dados com pares de chaves e valores, a chave é única e através dela é possível obter os valores armazenados;

• SortedList – Uma coleção que armazena os dados com pares de chaves e valores, onde os dados estão ordenados e além de acessar os valores através da chave, é possível acessá-los através do índice.

Agora que vimos um pouco sobre coleções em C# vamos ver o que mudou com a entrada de Generics em relação à manipulação de coleções. As coleções genéricas estão no namespace System.Collections.Generic. Veremos as principais coleções genéricas e seus equivalentes similares:

• ArrayList -> List<T>

• Queue -> Queue<T>

• Stack -> Stack<T>

• HashTable -> Dictionary<K,V>

• SortedList -> SortedDictionary<K,V>

Note que temos a letra T entre os sinais de menor e maior após o nome da coleção. T é um nome dado para o parâmetro, é uma prática comum utilizar a letra T, nos casos de Dictionary<K,V> e SortedDictionary<K,V> utiliza-se K de Key e V de value como prática de uso, mas tanto o T, K e V não são nomes obrigatórios para parâmetro de tipo. O Type Parameter ou Type Placeholder é a base de qualquer tipo genérico. A letra T é de Type, é com ele que o tipo será representado e que será informado quando criamos uma instância dele.

As coleções genéricas vão receber como parâmetro o tipo do objeto, assim se tornando fortemente tipadas. O Type Parameter vai receber o tipo que é tratado dentro da classe com a mesma referência. Para entender melhor, vamos ver um exemplo, imagine uma coleção do tipo List<T> que é similar ao ArrayList, coleção não genérica. O List<T> permite criar uma coleção genérica, para criar uma coleção de elementos do tipo Double, chamada lista, com o List<T> podemos usar a seguinte sintaxe:

```
List<double> lista = new List<double>();
```

Note que definimos o parâmetro como sendo um tipo Double, ou seja, a coleção “lista” agora somente aceitará elementos do tipo Double, se tentarmos inserir um elemento diferente, um erro será informado no Visual Studio, em tempo de compilação.

ArrayList vs List<T>

Vamos dar uma olhada agora em coleções com Generics na prática, nas próximas linhas veremos um exemplo das coleções genéricas citadas. Utilizarei o Visual Studio 2010, atualizado com o framework 4.0, mas os exemplos funcionam no Visual Studio 2008.

Primeiro vamos ver a criação de uma coleção não genérica do tipo ArrayList, e depois vamos comparar com a versão genérica List<T>. Crie um projeto C# no Visual Studio 2010 do tipo Windows Forms, adicione um botão na tela e no evento click adicione o código da **Listagem 1**.

No exemplo criamos uma coleção do tipo ArrayList chamada lista, em seguida inserimos cinco elementos do tipo int. Na linha 13 iniciamos um comando for para percorrer a coleção. Na linha 15 temos a conversão forçada, estamos aqui extraindo o conteúdo do ArrayList e convertendo em um tipo int. Na linha abaixo fazemos um cálculo simples, somando todos os itens na coleção para no final apresentar na tela. Note que na linha 8 temos um comentário, se removermos a //, o sistema vai compilar normalmente, e somente aparecerá um erro em tempo de execução.

Listagem 1. Coleção com ArrayList

```
1    ArrayList lista = new ArrayList();
2
3    lista.Add(10);
4    lista.Add(20);
5    lista.Add(30);
6    lista.Add(40);
7    lista.Add(50);
8    //lista.Add("A");
10
11  int total=0;
12
13  for (int i = 0; i <= lista.Count-1; i++)
14  {
15    int valor = (int) lista[i];
16
17    total += valor;
18
19  }
20
21  MessageBox.Show(total.ToString());
```

Lembre-se de adicionar o namespace System.Collections.

Agora vamos criar o mesmo exemplo utilizando uma coleção com Generics. Arraste outro botão na tela, e adicione o código na **Listagem 2**. A diferença aqui está na criação da coleção utilizando o List<T> no lugar do ArrayList. Temos um Type Parameter (T) que receberá um tipo, no exemplo utilizamos o int. Na linha 14 estamos resgatando o valor e inserindo em uma variável do tipo int.

Note que desta vez não houve a necessidade de realizar uma conversão forçada, como a coleção já tem um tipo definido, foi possível passar diretamente o valor para o tipo int.

Listagem 2. Coleção com List<T>

```
1 List<int> lista = new List<int>();
2
3 lista.Add(10);
4 lista.Add(20);
5 lista.Add(30);
6 lista.Add(40);
7 lista.Add(50);
8 //lista.Add("A"); //erro no VS
9
10  int total = 0;
11
12  for (int i = 0; i <= lista.Count - 1; i++)
13  {
14        int valor = lista[i]; // sem box e unboxing
15
16        total += valor;
17
18   }
19
20  MessageBox.Show(total.ToString());
```

Stack<T>

Uma coleção do tipo Stack<T> permite criar um grupo de dados “empilhados”. Imagine que você tenha diversos objetos empilhados um em cima do outro, quando você for pegar um objeto para usar, o comum é pegar o objeto de cima até chegar ao último. É dessa forma que o Stack trabalha, o primeiro item que ele disponibiliza é o último inserido na pilha, seguindo o princípio LIFO (Last In - First Out). Na **Listagem 3** criamos uma coleção do tipo Stack<int> que vai receber somente tipos int, em seguida temos um for que insere os elementos na pilha, um a um, através do método Push(), passando o número como parâmetro. Na linha 12 temos um while, que irá percorrer a coleção enquanto ela for maior que zero. Através do método Pop() obtemos os itens na ordem LIFO, ou seja, inserimos na ordem 10, 20, 30 e 40 e obtemos os números na ordem 40, 30, 20 e 10.

Listagem 3. Coleção com Stack<T>

```
1  Stack<int> pilha = new Stack<int>();
2
3  int[] numeros = new int[] { 10, 20, 30, 40 };
```

```
4
5  foreach (int num in numeros)
6  {
7      MessageBox.Show("Inserindo na pilha o item: " + num.ToString());
8      pilha.Push(num);
9
10 }
11
12 while (pilha.Count > 0)
13 {
14    int num = pilha.Pop();
15    MessageBox.Show("Obtendo o item: " + num.ToString());
16
17 }
```

Queue<T>

A coleção Queue coloca os elementos em uma fila. O mesmo exemplo usado na coleção Stack<T> foi recriado na **Listagem 4**, nela temos uma coleção do tipo Queue<int>, que somente vai aceitar elemento do tipo int. Dentro do foreach, na linha10, os elementos foram inseridos na fila através do método Enqueue, que recebe como parâmetro um número inteiro. Em seguida temos um comando while, percorrendo a fila. Na linha 17, através do método Dequeue, obtemos os elementos da fila, um a um, e apresentamos na tela. Para entender melhor, imagine uma fila de pessoas para serem atendidas por ordem de chegada, da mesma forma funciona a coleção Queue, quando inserimos na fila, os itens estavam na ordem 10, 20, 30 e 40 e quando vamos obter os itens, eles aparecem na ordem 10, 20, 30 e 40.

Listagem 4. Coleção com Queue<T>

```
1 Queue<int> fila = new Queue<int>();
2
3 int[] numeros = new int[] { 10, 20, 30, 40 };
4
5 foreach (int num in numeros)
6 {
7     MessageBox.Show("Inserindo na fila o item: " + num.ToString());
8
9
10    fila.Enqueue(num);
11
12
13 }
14
15 while (fila.Count > 0)
```

```
16 {
17   int num = fila.Dequeue();
18
19   MessageBox.Show("Obtendo um item na fila: " + num.ToString());
20
21 }
```

Dictionary<K,V>

A Dictionary cria uma coleção de dados que possui uma chave ligada ao valor que será armazenado na coleção. O nome K é a chave (Key), o V é o valor (Value). Na **Listagem 5** temos a criação de uma coleção do tipo Dictionary<K,V>, chamada telefones, nela vamos adicionar o nome de uma pessoa como chave (Key) e um telefone como valor. Da linha 4 a 6 estamos realizando esta tarefa. Na linha 11 estamos apresentando o valor do item que contém a chave “Tadashi”, já na linha 15 temos um foreach que percorre todos os elementos e apresenta o valor de cada um através de suas chaves correspondentes. Na linha 19 temos outro foreach percorrendo e apresentado os itens da coleção através dos valores.

Listagem 5. Coleção com Dictionary<K,V>

```
1 Dictionary<string, string> telefones =
2   new Dictionary<string, string>();
3
4 telefones.Add("Tadashi", "1111-1111");
5 telefones.Add("Sato", "2222-2222");
6 telefones.Add("Alexandre", "3333-3333");
7
8  //Pega o telefone através do nome (chave)
9
10
11 MessageBox.Show(telefones["Tadashi"]);
12
13
14 //Mostra todos os telefones através da chave
15 foreach (string chave in telefones.Keys)
16         MessageBox.Show(telefones[chave]);
17
18 //Mostra todos os telefones através do valor
19 foreach (string value in telefones.Values)
20         MessageBox.Show(value);
```

SortedDictionary<K,V>

A coleção SortedDictionary<K,V> é muito semelhante ao Dictionary<K,V>, nela também podemos armazenar os dados com pares de chaves e valores, mas o grande diferencial é que a lista é

ordenada, o que é necessário em alguns casos. É possível também acessar os valores através do índice. Um exemplo de SortedDictionary<K,V> pode ser visto na **Listagem 6**, onde criamos uma coleção SortedDictionary<string,string>, ou seja, tanto a chave quanto o valor serão do tipo string. Logo em seguida adicionamos nomes e telefones, da mesma forma que fizemos no exemplo de Dictionary<K,V>. Na linha 9 temos o foreach que apresenta os elementos da coleção em ordem alfabética de chaves.

Listagem 6. Coleção com SortedDictionary<K,V>

```
1    SortedDictionary<string, string> telefones =
2     new SortedDictionary<string, string>();
3
4    telefones.Add("Tadashi", "1111-1111");
5    telefones.Add("Sato", "2222-2222");
6    telefones.Add("Alexandre", "3333-3333");
7
8
9     foreach (string s in telefones.Keys)
10    {
11        MessageBox.Show( s);
12    }
```

Coleção personalizada com Generics

Neste último exemplo do artigo vamos ver como criar uma coleção personalizada para conhecer um pouco mais sobre o uso de Generics em coleções. Vamos criar uma classe que vai implementar uma interface também genérica. Também podemos utilizar generics de diversas formas, inclusive em interfaces, apesar do foco do artigo ser o uso de Generics em coleções. A interface que a classe vai implementar é a IEnumerable. Com a interface implementada, ganhamos o suporte a iterações em nossa classe, podendo, por exemplo, ser percorrida em um foreach, assim como os exemplos que fizemos no artigo.

Antes de iniciar a criação da coleção personalizada, vamos criar algumas classes de apoio, a primeira é a classe Pessoa (**Listagem 7**), que contém somente uma propriedade chamada nome do tipo string. Os seus métodos get e set permitem recuperar e inserir um valor na propriedade. O método ToString() que sobrescrevemos com a palavra reservada override faz com que a classe apresente a propriedade nome.

Listagem 7. Classe Pessoa

```
1 namespace EasyNet
2 {
3    public class Pessoa
4    {
5        private string nome;
```

```
6
7         public string Nome
8         {
9             get { return nome; }
10             set { nome = value; }
11         }
12
13         public override string ToString()
14       {
15           return nome;
16       }
17
18   }
19
20}
```

Vamos criar mais duas classes de apoio, Contato (**Listagem 8**) e Funcionario (**Listagem 9**), ambas herdam a classe Pessoa e contêm um construtor vazio.

Listagem 8. Classe Contato

```
1 namespace EasyNet
2 {
3    public class Contato : Pessoa
4    {
5
6        public Contato() { }
7
8    }
9 }
```

Listagem 9. Classe Funcionario

```
1 namespace EasyNet
2 {
3    public class Funcionario : Pessoa
4    {
5        public Funcionario() { }
6
7    }
8 }
```

Na **Listagem 10** temos a coleção genérica propriamente dita, chamada de ColecaoGenerics. Vamos dar uma olhada na linha 8, ela apresenta a sintaxe:

```
public class ColecaoGenerics<T> : IEnumerable<T>
```

Estamos definindo com isso uma classe genérica, onde o T (Type) que está após o nome da classe pode receber qualquer tipo. Após os ":" temos a interface IEnumerable<T> genérica, que vai nos oferecer o suporte para percorrer a coleção.

Se por algum motivo você quiser restringir que o tipo informado em T seja somente pessoa, você poderá usar a cláusula Where no final da linha, por exemplo:

```
ColecaoGenerics<T> : IEnumerable<T> where T : Pessoa
```

Dessa forma somente elementos do tipo Pessoa ou que herdam dela podem ser passados como parâmetro para criar a classe. O Where pode ser utilizado de diversas formas, para forçar que o elemento passado tenha um construtor, para restringir o conteúdo passado etc.

Na linha 11 foi criada uma Collection List<T> com o nome de lista. Na linha 13 temos um método chamado Retorna que retorna um tipo genérico T, que está dentro da coleção, o mesmo que é passado como parâmetro para a classe. Na linha 16 temos o método Adiciona, que recebe um objeto do tipo T e adiciona na lista. Na linha 19 e 22 temos dois métodos que não utilizaremos no artigo, mas que servem para limpar e contar os elementos da lista.

Listagem 10. Coleção Genérica

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace EasyNet
7 {
8    public class ColecaoGenerics<T> : IEnumerable<T>
9    {
10
11       private List<T> lista = new List<T>();
12
13       public T Retorna(int posicao)
14       { return lista[posicao]; }
15
16       public void Adiciona(T valor)
17       { lista.Add(valor); }
18
19       public void LimpaColecao()
20       { lista.Clear(); }
21
22       public int TotalItens
```

```
23        { get { return lista.Count; } }
24
25        #region IEnumerable<T> Members
26
27        // IEnumerable<T> extende IEnumerable, sendo
28        // necessário implementar duas versões
29        // de GetEnumerator().
30        public IEnumerator<T> GetEnumerator()
31        {
32             return lista.GetEnumerator();
33        }
34
35        #endregion
36
37        #region IEnumerable Members
38
39        System.Collections.IEnumerator
40            System.Collections.IEnumerable.GetEnumerator()
41        {
42            return lista.GetEnumerator();
43        }
44
45        #endregion
46    }
47 }
```

Na **Listagem 11** vamos testar a classe ColecaoGenerics<T>. Adicione um botão em sua tela Windows Forms e adicione o código, depois adicione um elemento ListBox. Na primeira linha criamos uma coleção personalizada com a classe ColecaoGenerics, passando um objeto do tipo Pessoa como parâmetro, em seguida adicionamos três objetos do tipo Contato na coleção e um objeto do tipo Funcionário. Como ambos herdam da classe Pessoa, a coleção aceita a adição dos elementos. Na linha 20 temos um foreach que consegue percorrer a classe, pois implementamos a interface IEnumerable<T>, e carrega o conteúdo dos itens para o ListBox.

Listagem 11. Testando a Coleção Genérica

```
1            ColecaoGenerics<Pessoa> lista =
2                new ColecaoGenerics<Pessoa>();
3
4            Contato contato1 = new Contato();
5            contato1.Nome = "Alexandre";
6            lista.Adiciona(contato1);
7
8            Contato contato2 = new Contato();
```

```
9            contato2.Nome = "Tadashi";
10            lista.Adiciona(contato2);
11
12            Contato contato3 = new Contato();
13            contato3.Nome = "Sato";
14            lista.Adiciona(contato3);
15
16            Funcionario funcionario1 = new Funcionario();
17            funcionario1.Nome = "Claudia";
18            lista.Adiciona(funcionario1);
19
20            foreach (Pessoa itens in lista)
21            {
22                listBox1.Items.Add(itens);
23            }
24
```

Conclusão

Generics não é uma novidade, linguagens de programação antigas já utilizavam o conceito. Neste artigo vimos somente uma introdução ao recurso de Generics, mas já é possível ver o seu poder na construção de projetos. Ainda existe muito a estudar para conhecer todo o potencial de Generics, assim como o seu uso com métodos, classes, interfaces e structs.

No artigo vimos com utilizar Generics na manipulação de coleções, vimos o conceito de Box e Unbox e comparamos os benefícios de utilizar uma coleção genérica em vez de não genérica, por fim, vimos uma classe que cria uma coleção personalizada usando de Generics.

Conhecer e aperfeiçoar o uso de Generics auxilia na criação de aplicativos mais enxutos e flexíveis, podendo reduzir consideravelmente o uso de códigos redundantes, principalmente devido à resolução dos problemas de conversões de tipos que um elemento genérico pode proporcionar. Esses são somente alguns dos benefícios que os Generics apresentam, o assunto pode parecer complexo para quem está iniciando no desenvolvimento de softwares, mas com certeza é importante e hoje parte fundamental da evolução da plataforma .net.

Links

Blog do autor
http://alexandretadashi.net/

Twitter – Alexandre Tadashi
http://twitter.com/atsh2