GENERICS – PARTE II

Anteriormente foi apresentado o Generics. Vimos que se trata de um recurso que permite criar templates de objetos com flags que determinarão tipos somente no momento da compilação. E criamos a classe Envelope:

```
public class Envelope<T>
{
        private string destinatario;
        private T conteudo;
        public string Destinatario
        {
            get { return destinatario; }
            set { destinatario = value; }
        }
        public T Conteudo
        {
            get { return conteudo; }
            set { conteudo = value; }
        }
}
```

Que endereça um objeto do tipo T ao destinatário.

Definindo T's adicionais

O "T" no subtítulo foi apenas ilustrativo... T é apenas um nome que damos para identificarmos o tipo definido na hora de consumir uma classe que implemente Generics, e você pode criar quantos nomes precisar. Por exemplo, tomemos nossa classe Envelope e digamos que queiramos flexibilizar o uso do membro destinatário...

```
public class Envelope<T, D>
{
        private D destinatario;
        private T conteudo;
        public D Destinatario
        {
            get { return destinatario; }
            set { destinatario = value; }
        }
        public T Conteudo
        {
            get { return conteudo; }
            set { conteudo = value; }
        }
}
```

Desta forma você consegue "generalizar" mais de um tipo. Mas o que acontece se você quiser garantir que essa generalização não seja flexível demais?

Limitando os T's

E se eu não quiser exatamente que todas as classes do framework, indiscriminadamente, sejam usadas no meu objeto? Como garantir que minha equipe está usando minha classe de uma forma ideal. A resposta está na cláusula where.

```
public class Envelope<T, D>
        where T : IMessage
        where D : IContact
{
        private D destinatario;
        private T conteudo;
        public D Destinatario
        {
            get { return destinatario; }
            set { destinatario = value; }
        }
        public T Conteudo
        {
            get { return conteudo; }
            set { conteudo = value; }
        }
}
```

Com o where podemos limitar que classes serão aceitas no template. Você pode estar se perguntando "qual a vantagem de usar Generics se o template já está sendo tipado?". Eu te respondo: engano seu, o template não está sendo tipado. Mas ele vai receber um número limitado de tipos. Existem situações muito específicas onde esse conhecimento será muito útil.

System.Collections.Generic.Dictionary<>

Entre as classes do System.Collection.Generic, Dictionary<> é a que chega mais perto do já conhecido e útil HashTable. O template de Dictionary<> aceita dois tipos, a saber: TKey é o tipo a ser usado para a chave que irá identificar os objetos dentro do Dictionary, e TValue é o tipo dos valores que o Dictionary irá armazenar.

```
Pessoa pessoa = new Pessoa();
Dictionary<string, Pessoa> registros = new Dictionary<string, Pessoa>();
registros[pessoa.Cpf] = pessoa;
Pessoa pessoa2 = registros[pessoa.Cpf];
```

Observe, novamente, que para resgatar o objeto de dentro do Dictionary para pessoa2, não foi necessário fazer o tradicional Cast.

System.Collections.Generic.Queue<> e Stack<>

Outras duas classes úteis: Queue<> e Stack<> que respectivamente operam os conhecidos FIFO e LIFO, respectivamente. Queue<> enfileira objetos do tipo T e os devolve na ordem de chegada. Stack<> empilha os objetos do tipo T e os devolve sempre começando pelo último empilhado e terminando pelo primeiro empilhado.

```
Queue<Pessoa> fila = new Queue<Pessoa>();
fila.Enqueue(new Pessoa());
fila.Enqueue(new Pessoa());
Pessoa pessoa = fila.Dequeue();

Stack<Pessoa> pilha = new Stack<Pessoa>();
pilha.Push(new Pessoa());
pilha.Push(new Pessoa());
Pessoa pessoa2 = pilha.Pop();
```

Usando Generics em Interfaces

Generics não é um recurso restrito às classes. Interfaces também podem utilizá-lo para se tornarem genéricas. Um bom exemplo é a interface System.Collections.Generic.IList<>. É possível implementar esta interface quando se pretende criar suas próprias coleções. Um bom exemplo disso é precisar criar um List que dispare eventos quando objetos são inseridos ou removidos de sua coleção.

Um exemplo simples de uma interface com Generics:

```
public interface IContainer<T>
{
        void Maintain(T Item);
        T Obtain();
}
```

Exemplo de utilização:

```
public class ContainerPessoa : IContainer<Pessoa>
{

        private Pessoa pessoa;

        public void Maintain(Pessoa Item)
        {
            pessoa = Item;
        }

        public Pessoa Obtain()
        {
            return pessoa;
        }

}
```

Usando Generics em Delegates

A utilização de Generics em delegates também cria possibilidades muito interessantes. De fato, existem muitos delegates no framework que usam este artifício. Um deles é o delegate System.Comparison<> que recebe um tipo T em seu template para métodos que recebam dois objetos do tipo T e retornam um int, normalmente para fazer comparações. Este delegate é utilizado pelo método List<>.Sort que classifica

todos os itens de um List baseado nos critérios de ordenação determinado pelo método Comparison<> delegado.

Para fazer um exemplo, na classe pessoa criaremos dois métodos:

```
public static int ComparaNascimentoAsc(Pessoa p1, Pessoa p2)
{
        if (p1.DataNascimento > p2.DataNascimento)
                return 1;
        else
                return -1;
}
```

```
public static int ComparaNascimentoDesc(Pessoa p1, Pessoa p2)
{
        if (p1.DataNascimento > p2.DataNascimento)
                return -1;
        else
                return 1;
}
```

Os dois métodos obedecem à assinatura estabelecida pelo System.Comparison<Pessoa>, o que muda entre eles é o critério: O primeiro retorna 1 se o primeiro objeto passado for o maior, o segundo retorna -1 se o primeiro objeto foi o maior. Estes valores de retorno obedecem a um padrão de valores para classificação de objetos definido no framework, Existe um terceiro valor “0” para quando os dois objetos forem iguais, mas não possui utilidade no nosso exemplo, embora seria útil para criar novos critérios.

Tendo um List<Pessoa> é possível ordená-lo usando o método sort, delegando um dos dois métodos que acabamos de criar, de acordo com a necessidade. Assim sendo, para ordenar de forma ascendente:

```
pessoas.Sort(Pessoa.ComparaNascimentoAsc);
```

Do contrário, para ordenação descendente:

```
pessoas.Sort(Pessoa.ComparaNascimentoDesc);
```

Considerações Finais

Obviamente tem muito mais a saber sobre a utilização de Generics em livros e outros materiais. O namespace System.Collections.Generic está recheado de classes, interfaces e muito mais para pronta utilização e se lá não existir o que você precisa, você já pode se arriscar a implementar por si mesmo.

Em breve voltarei com novos artigos para o .Net. Até lá.