Material C# Generics

Sintaxe

Classes

```
public class ClasseGenerica<T>{}
```

Interfaces

```
public interface IInterfaceGenerica<T>{}
```

Oque isso significa?

Classes genéricas são classes que recebem tipos (*classes*) para seus métodos trabalharem, isso implica que na prática podemos ter métodos que aceitam vários tipos ao invés de gerar classes especificas para tratar cada tipo de dados.

Exemplo List< T >

```
using System.Collections.Generic;

public sealed class Produto{
    public string Nome{get; private set;}
    public decimal Preco{get; private set;}

    public Produto(string nome, decimal preco){
        Nome = nome;
        Preco = preco;
    }
}

public sealed class Pedido{
    public List<Produto> Produtos;

    public Pedido(){
        Produtos = new List<Produto>();
    }
}
```

Você já deve ter utilizado a classe genérica List do C# e notado que idependente do tipo que você coloca nela o comportamento dos métodos da classe List é o mesmo. Na prática imagina o quão complexo seria de para cada tipo T você tivesse que fazer uma implementação do método **Add** que faz um simples insert na lista, isso seria um caos kkkkkk. Portanto podemos concluir que as classes genéricas servem para que possamos ter o reuso do nosso código, porém é importante ter em mente antes de sair usando generics para todo lado que sua classe ou método deve ser de fato algo genérico, voltemos ao método **Add** de fato ele é um método que faz sentido generalizar e indo mais além na abstração olhe a classe **List**, tente pensar em um tipo que não pode formar uma coleção e avalie se a generalização de **List** é uma boa ideia.

Exemplos Práticos

Antes de entrar nesse exemplo prático gostaria de apresentar um conceito de filtro para nossas generalizações, você deve estar pensando:

por qual motivo eu faria um filtro em uma generalização se meu objetivo é generalizar ? (kkkkkkk)

Bom imagine que você tem dois tipos que precisa tratar de mesma maneira, porém eles não estão relacionados por uma interface, nem por uma classe abstrata e também não tem não se relacionam por herança, sim são casos muito especificos, então bora ver o conceito de constrain e alguns exemplos em seguida.

Constrains

É importante dizer que esse assunto é extenso e que essas constrains podem te ajudar e muito a definir o conjunto correto de tipos que você deseja processar por isso vou deixar um link com todas elas.

Restrições a parâmetros de tipo – Guia de Programação em C# | Microsoft Docs

Nesse material vou tratar de dois tipos de constrains, a mais simples onde definimos uma obrigatoriedade de herança e outro caso onde pedimos que o tipo **T** implemente uma ou mais interfaces.

Herança Obrigatória

```
public abstract class MinhaClasse {}

public interface IInterfaceGenerica<T> where T: MinhaClasse {
    void FazerAlgumaCoisa();
}

public class ClasseGenerica<T> where T: MinhaClasse         {
    void FazerAlgumaCoisa(){};
}
```

Acima temos uma interface genérica que obriga que o tipo **T** seja uma seja uma subclasse de **MinhaClasse**, pense se isso faz algum sentido, se você não for capaz de opinar sobre isso agora ta tudo certo, o objetivo é que até o final do material você seja capaz de dizer se isso é uma boa ideia ou não.

Implementações Obrigatórias

```
interface IMinhaInterface {}
interface IMinhaSegundaInterface {}

public class ClasseGenerica<T>
where T: IMinhaInterface, IMinhaSegundaInterface
{
   void FazerAlgumaCoisa(){};
}

public interface IInterfaceGenerica<T>
where T: IMinhaInterface, IMinhaSegundaInterface
{
    void FazerAlgumaCoisa();
}
```

Primeiro exemplo:

Vamos definir uma classe abstrata Model

```
public abstract class Model{
    public Guid Id {get; private set;}
    public DateTime CreatedAt {get; private set;}
    public DateTime UpdatedAt {get; private set;}

    public Model(){
        Id = new Guid();
        CreatedAt = DateTime.Now;
        UpdatedAT = CreatedAt;
    }
}
```

Agora vamos definir alguns modelos

```
public sealed class Carro : Model{
    public string Nome {get; private set;}
    public string Placa {get; private set;}

    public Carro(string name, string placa){
        Name = name;
        Placa = placa;
    }
}

public sealed class Cachorro: Model{
    public string Nome {get; private set;}

    public Cachorro(string name, string placa){
        Name = name;
    }
}
```

Definindo a interface genérica para os repositórios

```
using System.Collections.Generic;

public interface IGenericRepository<T> where T: Model{
    void Insert(Guid id);
    List<T> Find();
    T Find(Guid id);
    void Update(Guid id, T model);
    void Delete(Guid id);
}
```

Pense no porque essa interface pode ser genérica.

Nesse exemplo não serão definidas as implementações dessas interfaces, quero que a atenção fique no uso de generics.

Agora vamos construir duas interfaces para que possamos manipular especificamente as Models: Carro e Cachorro

```
public interface ICarroRepository : IGenericRepository<Carro>
{
    Carro FindByPlaca(string placa);
}

public interface ICachorroRepository : IGenericRepository<Cachorro>
{
    List<Cachorro> FindByNome(string nome);
}
```

Galera a gente acabou de economizar dedo aqui **kkkkk**, agora nossas interfaces **ICarroRepository** e **ICachorroRepository** já carregam todos os métodos de **IGenericRepository** com seus devidos **Models** especificados. Com isso podemos pensar em definir somente os métodos especificos de cada repositório, perceba que **ICarroRepository** tem lá a assinatura do método **FindByPlaca(string placa)** e **ICachorroRepository** tem **FindByNome(string nome)**.

Espero que você tenha conseguido entender o assunto Generics aplicado no C#, abaixo vou deixar alguns links uteis sobre esse assunto.

C# - Revendo conceitos sobre Generics (macoratti.net)

Classes e métodos genéricos | Microsoft Docs

Feito por um fã de ursinhos carinhosos S2.