



Fundamentos de C#
Generics

Softblue
cursos online

Tópicos Abordados

- Generics
 - Problemas de não usar o generics
 - Boxing e Unboxing
 - Tipos de dados incorretos
- Generics e classes
- Generics e structs
- Múltiplos tipos
- Valor padrão
- Generics constraints
- Herança e implementação com generics
- Generics e métodos
 - Sobrecarga e sobrescrita

Generics

- O generics permite que classes e métodos utilizem tipos parametrizados
 - A mesma classe ou método pode receber dados dos mais variados tipos

Exemplo sem Generics



- Suponha que você queira criar uma classe para agrupar elementos que aceite vários tipos

```
class Pacote
{
    public object Elem1 { get; set; }
    public object Elem2 { get; set; }
    public object Elem3 { get; set; }

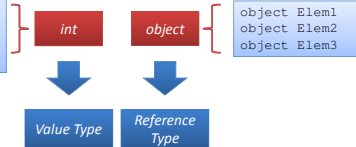
    public void Mostrar()
    {
        Console.WriteLine("Elemento 1: " + Elem1.ToString());
        Console.WriteLine("Elemento 2: " + Elem2.ToString());
        Console.WriteLine("Elemento 3: " + Elem3.ToString());
    }
}
```

O tipo **object** é usado por ser genérico

Problema #1: Boxing e Unboxing



```
Pacote p = new Pacote();
p.Elem1 = 10;
p.Elem2 = 20;
p.Elem3 = 30;
p.Mostrar();
```



```
object Elem1 = 10;
```

Um novo objeto é criado no managed heap, e o valor é copiado para dentro dele

Boxing

```
int i = (int) p.Elem1;
```

O dado é copiado para a stack e o objeto deixa de ser referenciado

Unboxing

Boxing e unboxing causam problemas de performance

Problema #2: Tipo de Dado Incorreto



- Suponha que você queira colocar apenas strings no pacote

```
Pacote p = new Pacote();
p.Elem1 = "abc";
p.Elem2 = true;
p.Elem3 = 10.4;
```

Nada impede que outros tipos sejam fornecidos

```
string s1 = (string)p.Elem1;
string s2 = (string)p.Elem2;
string s3 = (string)p.Elem3;
```

Erro na execução
(InvalidCastException)

Não existe uma proteção com relação ao tipo usado

Generics e Classes



- O uso do generics resolve ambos os problemas
 - Evita o boxing e unboxing
 - Verifica os tipos de dados na compilação

```
class Pacote<T>
{
    public T Elem1 { get; set; }
    public T Elem2 { get; set; }
    public T Elem3 { get; set; }
}
```

A classe usa generics, e o tipo é representado por T

O tipo do dado assume o tipo T

Instanciação com o Uso de Generics



- Quando a classe usa generics, o tipo parametrizado deve ser fornecido no momento da criação do objeto

```
Pacote<string> p = new Pacote<string>();
```

O tipo T será substituído por string

```
p.Elem1 = "abc";
```

Boxing inexistente

```
p.Elem2 = true;
p.Elem3 = 10.4;
```

Não compila

```
string s1 = p.Elem1;
```

Unboxing inexistente

Generics e Structs




- *Structs* também podem se beneficiar do uso de generics

```
struct Triangulo<T>
{
    private T l1;
    private T l2;
    private T l3;

    public Triangulo(T l1, T l2, T l3)
    {
        this.l1 = l1;
        this.l2 = l2;
        this.l3 = l3;
    }
}
```

```
Triangulo<int> t1 = new Triangulo<int>(2, 3, 4);
Triangulo<double> t2 = new Triangulo<double>(3.5, 2.5, 9.0);
```

Múltiplos Tipos


- Mais de um tipo de dado é suportado pelo generics

2 tipos parametrizados

```

class Propriedade<TChave, TValor>
{
    private TChave chave;
    private TValor valor;


    public Propriedade(TChave chave, TValor valor)
    {
        this.chave = chave;
        this.valor = valor;
    }
}

```

```

Propriedade<int, string> p = new Propriedade<int, string>(1, "ABC");

```

Valor Padrão


- Generics podem ser usados com *reference types* e *value types*
 - O valor padrão de *value types* é 0
 - O valor padrão de *reference types* é null
- A palavra-chave **default()** pode ser usada

```

class MyClass<T>
{
    public T Retonar()
    {
        T temp = default(T);
        return temp;
    }
}

```

```

MyClass<int> m1 = new MyClass<int>();
m1.Retonar();

```


0

```

MyClass<string> m2 = new MyClass<string>();
m2.Retonar();

```

null

Generics Constraints


- O generics permite restringir os tipos de dados que podem ser usados de acordo com algumas regras

Constraint	Restrição
where T : struct	<T> deve ser um <i>value type</i>
where T : class	<T> deve ser um <i>reference type</i>
where T : IBaseInterface	<T> deve implementar a interface <i>IBaseInterface</i>
where T : BaseClass	<T> deve herdar da classe <i>BaseClass</i>
where T : new()	<T> deve ter um construtor sem parâmetros

Exemplos de Constraints

```
class MyClass<T> where T : struct { }
```

<T> deve ser um *value type*

```
class MyClass<T> where T : IMyInterface { }
```

<T> deve implementar *IMyInterface*

```
class MyClass<T> where T : new() { }
```

<T> deve ser ter um construtor sem parâmetros

```
class MyClass<T> where T : class, IMyInterface, new() { }
```

<T> deve ser um *reference type*, implementar *IMyInterface* e ter um construtor sem parâmetros

```
class MyClass<T, V> where T : BaseClass where V : new() { }
```

<T> deve ser uma subclasse de *BaseClass* e <V> deve ter um construtor sem parâmetros

Generics e a Herança/Implementação

- Uma classe pode herdar de outra que usa generics, ou implementar uma interface definida com generics

```
interface IPrintable<T>
{
    void Print(T info);
}
```

```
class MyClass<T> : IPrintable<T>
{
    public void Print(T info)
    {
        Console.Write(info);
    }
}
```

MyClass usa generics

Generics e a Herança/Implementação

- Uma classe não precisa, obrigatoriamente, usar o generics definido na sua superclasse ou superinterface

```
interface IPrintable<T>
{
    void Print(T info);
}
```

```
class MyClass : IPrintable<int>
{
    public void Print(int info)
    {
        Console.Write(info);
    }
}
```

MyClass não usa generics

O tipo <T> deve ser especificado

Generics com Métodos

- Não são apenas classes que podem se beneficiar do uso do generics
 - Métodos também podem

```

class Utils
{
    public void SwapValues<T>(ref T v1, ref T v2)
    {
        T temp = v1;
        v1 = v2;
        v2 = temp;
    }
}

```

A classe não precisa usar generics

O uso do generics é definido no nome do método

Generics com Métodos

```

int i = 3;
int j = 7;
Utils u = new Utils();
u.SwapValues<int>(ref i, ref j);

```

O tipo de dado é fornecido na chamada ao método

```

u.SwapValues(ref i, ref j);

```

O tipo pode ser omitido se o compilador consegue inferi-lo

Generics também podem ser usados em métodos estáticos

Métodos Estáticos com Generics

- Métodos estáticos também podem usufruir dos benefícios do generics

```

class Utils
{
    public static void SwapValues<T>(ref T v1, ref T v2) { }
}
Utils.SwapValues<int>(ref i, ref j);

```

```

class Utils<T>
{
    public static void SwapValues(ref T v1, ref T v2) { }
}
Utils<int>.SwapValues(ref i, ref j);


```

```

class Utils<V>
{
    public static void SwapValues<T>(ref T v1, ref T v2) { }
}
Utils<char>.SwapValues<int>(ref i, ref j);

```

Constraints em Métodos




- Métodos que usam generics também podem ter constraints

```
public void SwapValues<T>(ref T v1, ref T v2) where T : struct
{ }
```

<T> deve ser um value type

Sobrescrita de Métodos



- Se um método é sobrescrito, ele deve conter os mesmo tipos parametrizados e constraints


```
class BaseClass
{
    public virtual void Print<T>(T info) where T : struct { }
}
class SubClass : BaseClass
{
    public override void Print<T>(T info) { }
```

<T> deve estar presente

As constraints se aplicam também ao método sobrescrito

Declarar constraints no método sobrescrito gera erro de compilação

Sobrecarga de Métodos



- É possível aplicar a sobrecarga em métodos definidos com generics
- É possível também ter métodos sobrecarregados que usam e não usam generics

```
public void Print<T>(T info) { }
public void Print(string info) { }
```

Sobrecarga

O método escolhido para execução vai depender do parâmetro passado

