# UPskill – Java+.NET

## Programação Orientada a Objetos – Genéricos

UPskill
Digital Skills & Jobs

P.PORTO

# INTRODUÇÃO

Os *Generics* em C# permitem a criação de classes, métodos e interfaces que funcionam com qualquer tipo de dado, oferecendo maior flexibilidade e segurança no desenvolvimento. Eles permitem evitar duplicação de código e fornecem segurança em tempo de compilação, assegurando que apenas tipos compatíveis sejam utilizados.

Vantagens do uso de *Generics*:

- Reutilização de Código: Permitem que uma única classe ou método funcione com diferentes tipos de dados, sem precisar de duplicação de código.
- Segurança de Tipos: Detetam erros de tipo em tempo de compilação, reduzindo o risco de exceções em tempo de execução.
- Desempenho: Evitam o *boxing* e *unboxing*, que ocorre com tipos de referência, melhorando a eficiência.

Em C#, as restrições de tipo genérico (ou limites) especificam os requisitos ou limitações para os parâmetros de tipo utilizados em genéricos. Essas restrições ajudam a garantir que os tipos passados para os genéricos possuam certas características, métodos ou interfaces, tornando o código mais robusto e funcional (ver Lista de limites).

Benefícios de usar limites:

- Segurança do Código: As restrições impedem que tipos inválidos sejam passados para classes ou métodos genéricos, garantindo que apenas tipos válidos sejam utilizados.
- Funcionalidade Aumentada: As restrições permitem o acesso a membros e funcionalidades específicas dos tipos, tornando os genéricos mais poderosos e flexíveis.
- Melhor Legibilidade e Manutenção: Ao definir restrições, estabelece-se expectativas claras sobre como os tipos genéricos devem comportar-se, tornando o código mais fácil de entender e manter.

Tipos de Restrições de Tipo (ver Lista de limites completa)

- where T : struct: Restringe o tipo T para ser um tipo de valor não nulo, ou seja, tipos como int, float, bool, structs personalizadas, etc.
- where T : class: O argumento T deve ser um tipo de referência.
- where T : class?: O argumento type deve ser um tipo de referência, anulável ou não anulável.
- where T : new(): O argumento type deve ter um construtor sem parâmetros público.
- where T : <classe base>: O argumento type deve ser ou derivar da classe base especificada.

- where T : IInterface: O argumento type deve ser ou implementar a interface especificada.

Nas demonstrações seguintes, utilizaremos os tipos de dados definidos em C#, bem como a classe Animal, conforme definida a seguir:

```
public class Animal {
    public string Nome { get; set; }
    public string Tipo { get; set; }
    public int Idade { get; set; }

    public Animal(string nome, string tipo, int idade) {
        Nome = nome;
        Tipo = tipo;
        Idade = idade;
    }

    public override string ToString() {
        return $"Nome: {Nome}, Tipo: {Tipo}, Idade: {Idade}";
    }
}
```

## EXERCÍCIO 1 – GENÉRICOS

Analise os seguintes exemplos sobre genéricos.

**Exemplo 1: Classe Genérica**

```
public class Caixa<T> {
    private T item;
    public void AdicionarItem(T item) {
        this.item = item;
    }
    public T ObterItem() {
        return item;
    }
}
```

Uso da Classe Genérica:

```
var caixaInt = new Caixa<int>();
caixaInt.AdicionarItem(42);
Console.WriteLine(caixaInt.ObterItem());

var caixaString = new Caixa<string>();
caixaString.AdicionarItem("Olá, Mundo!");
Console.WriteLine(caixaString.ObterItem());

var caixaAnimal = new Caixa<Animal>();
caixaAnimal.AdicionarItem(new Animal("Rex", "Cachorro", 5));
Console.WriteLine(caixaAnimal.ObterItem());
```

**Exemplo 2: Classe com Lista de *Generics***

```
public class Armazem<T> {
    private List<T> itens = new List<T>();

    public void AdicionarItem(T item) {
        itens.Add(item);
    }

    public List<T> ObterItens() {
        return itens;
    }
}
```

Uso da Classe Genérica:

```
var armazemNumeros = new Armazem<int>();
armazemNumeros.AdicionarItem(10);
armazemNumeros.AdicionarItem(20);

foreach (var numero in armazemNumeros.ObterItens()) {
    Console.WriteLine(numero); // Saída: 10, 20
}

var armazemNomes = new Armazem<string>();
armazemNomes.AdicionarItem("João");
```

```
armazemNomes.AdicionarItem("Maria");

foreach (var nome in armazemNomes.ObterItens()) {
    Console.WriteLine(nome);
}

var armazemAnimais = new Armazem<Animal>();
armazemAnimais.AdicionarItem(new Animal("Rex", "Cachorro", 5));
armazemAnimais.AdicionarItem(new Animal("Mia", "Gato", 3));

foreach (var animal in armazemAnimais.ObterItens()) {
    Console.WriteLine(animal);
}
```

**Exemplo 3: Método Genérico**

```
public class Utils {
    public static void Exibir<T>(T item) {
        Console.WriteLine(item);
    }
}
```

Uso do método Genérico:

```
Utils.Exibir<int>(42);
Utils.Exibir<string>("Olá, Mundo!");
Utils.Exibir<Animal>(new Animal("Rex", "Cachorro", 5));
```

## EXERCÍCIO 2 – LIMITES

Analise o seguinte exemplo sobre limites.

```
internal interface IImprimivel {
    void Imprimir();
}
```

```
public class Carro : IImprimivel {
    public string Marca { get; set; }
```

```
    public string Modelo { get; set; }
    public string Cor { get; set; }
    public int Ano { get; set; }

    public Carro(string marca, string modelo, string cor, int ano) {
        Marca = marca;
        Modelo = modelo;
        Cor = cor;
        Ano = ano;
    }

    public Carro() {
        Marca = "Sem marca";
        Modelo = "Sem modelo";
        Cor = "Sem cor";
        Ano = 0;
    }

    public override string ToString() {
        return $"Marca: {Marca}, Modelo: {Modelo}, Cor: {Cor}, Ano: {Ano}";
    }

    public void Imprimir() {
        Console.WriteLine(this);
    }
}
```

```
internal class Frota<T> where T : class, IImprimivel, new() {
    private readonly List<T> _itens = new List<T>();

    public Frota() { }

    public void Adicionar(T item) {
        _itens.Add(item);
    }
```

```
    public void Imprimir() {
        foreach (var item in _itens) {
            item.Imprimir();
        }
    }
    public void Imprimir(T item) {
        item.Imprimir();
    }

    public void Remover(T item) {
        _itens.Remove(item);
    }

    public static T CriarNovaInstancia() {
        T novoItem = new T();
        return novoItem;
    }

    public void AdicionarNovoItem() {
        T novoItem = CriarNovaInstancia();
        Adicionar(novoItem);
    }
}
```

Uso:

```
Frota<Carro> frota = new Frota<Carro>();
frota.Adicionar(new Carro("Chevrolet", "Cruze", "Preto", 2019));
frota.Adicionar(new Carro("Ford", "Fiesta", "Branco", 2018));
frota.Adicionar(new Carro("Fiat", "Uno", "Vermelho", 2017));

frota.Imprimir();

frota.AdicionarNovoItem();

frota.Imprimir();
```

## EXERCÍCIO 3 – REPOSITÓRIO DE DADOS

te exercício visa implementar um padrão de projeto simples, utilizando genéricos. O objetivo é criar um repositório genérico de dados, ou seja, uma estrutura para armazenar e manipular objetos de diferentes tipos, sem precisar escrever código duplicado para cada tipo de dado.

Crie uma classe genérica chamada Repositorio<T>, que simula um repositório de dados. A classe deve ter os seguintes métodos:

- Adicionar(T item): Este método deve adicionar um item do tipo T ao repositório.
- Procurar(int id): Este método deve procurar e retornar um item pelo seu id.. O método vai procurar na lista o item que possui o id que foi fornecido como parâmetro. Para isso, a classe T deve ter uma propriedade Id, por isso a restrição where T : IIdentificavel.
- Listar(): Este método deve listar todos os itens armazenados no repositório.
- Remover(int id): Este método deve remover um item do repositório, com base no id.

Restrições:

- **Onde T : class, IIdentificavel**: A restrição where T : class, IIdentificavel garante que qualquer tipo usado como T no repositório será uma classe (class) e implementará a interface IIdentificavel, que deve ter uma propriedade Id do tipo int. Isso é necessário para os métodos de Buscar() e Remover(), onde precisamos acessar o Id do item.
- **A interface IIdentificavel**: Esta interface deve ser simples, contendo apenas a propriedade Id:

```
public interface IIdentificavel {
    int Id { get; set; }
}
```

## EXERCÍCIO 4 - PILHA GENÉRICA COM OPERAÇÕES BÁSICAS

Implemente uma classe genérica chamada Pilha<T>, que simula uma estrutura de dados do tipo pilha. A classe deve ter:

- Empilhar(T item) - Adiciona um item ao topo da pilha.
- Desempilhar() - Remove e retorna o item do topo da pilha.
- Topo() - Retorna o item do topo sem removê-lo.
- Tamanho() - Retorna o número de itens na pilha.

Teste a classe criando uma pilha de números inteiros e uma pilha de strings.

## EXERCÍCIO 5 - MÉTODO GENÉRICO DE TROCA

Implemente uma classe genérica chamada Pilha<T>, que simula uma estrutura de dados do tipo pilha. Implemente um método genérico chamado Trocar<T>, que recebe dois parâmetros do tipo T e troca os seus valores. O método deve funcionar para qualquer tipo de dado. Teste o método com tipos como int, string e uma classe personalizada, como Pessoa (onde a pessoa tem nome e idade).