

Pedro Inácio Rodrigues Pontes

## **Prática 3, Conversor de Unidades de Medida**

Belo Horizonte, Brasil

2025

# 1 Introdução

O objetivo da presente prática foi produzir um conversor de unidades de medida em C# utilizando da interface gráfica Avalonia UI. Aqui estão os requerimentos para a interface:

1. Permitir a seleção do tipo de conversão através de um ListBox
2. Ter um campo de entrada para o valor de origem
3. Ter um campo de saída para o valor convertido
4. Incluir um botão para executar a conversão
5. Tipos de Conversão e Fórmulas

## 2 Desenvolvimento

Para conseguir alcançar o objetivo da confecção do conversor de unidades de medida, o projeto foi dividido utilizando o padrão MVVM (Model-View-ViewModel). View define a interface do usuário, onde é utilizado o código .xaml do avalonia, Model define as estruturas de dados da aplicação, ViewModel conecta os dois.

### 2.1 Model

Para a seção model, foi criada a classe `ConversionType`, a qual define a estrutura básica de uma conversão, que tem nome e sua fórmula, que foi representada utilizando uma *Delegate Property*, a qual consiste em uma propriedade de classe que representa uma função que recebe um valor de parâmetro e devolve outro.

```
namespace ConversorDeUnidadesDeMedida.Models
{
    public class ConversionType
    {
        public string Name { get; set; }
        public Func<double, double> ConversionFormula { get; set; }

        public ConversionType(string name, Func<double, double> conversionFormula)
        {
            Name = name;
            ConversionFormula = conversionFormula;
        }
    }
}
```

## 2.2 Services

Além da divisão base do modelo MVVM, foi adicionado também o modelo Services, onde se encontra o `ConversionService`, o qual retorna uma lista com todos os tipos de conversão:

```
namespace ConversorDeUnidadesDeMedida.Services
{
    public static class ConversionService
    {
        public static List<ConversionType> GetConversionTypes()
        {
            return new List<ConversionType>
            {
                new ConversionType("Celsius para Fahrenheit", c => (c * 1.8) + 32),
                new ConversionType("Fahrenheit para Celsius", f => (f - 32) / 1.8),
                new ConversionType("Celsius para Kelvin", c => c + 273.15),
                new ConversionType("Kelvin para Celsius", k => k - 273.15),
                new ConversionType("Metros para Pés", m => m * 3.28084),
                new ConversionType("Pés para Metros", ft => ft * 0.3048),
                new ConversionType("Quilômetros para Milhas", km => km * 0.621371),
                new ConversionType("Milhas para Quilômetros", mi => mi * 1.60934),
                new ConversionType("Quilogramas para Libras", kg => kg * 2.20462),
                new ConversionType("Libras para Quilogramas", lb => lb * 0.453592),
                new ConversionType("Gramas para Onças", g => g * 0.035274),
                new ConversionType("Onças para Gramas", oz => oz * 28.3495),
                new ConversionType("Litros para Galões", l => l * 0.264172),
                new ConversionType("Galões para Litros", gal => gal * 3.78541),
                new ConversionType("Mililitros para Onças Fluidas", ml => ml * 0.03),
                new ConversionType("Onças Fluidas para Mililitros", flOz => flOz *
            };
        }
    }
}
```

## 2.3 View

Para o modelo View, representado principalmente por `MainWindow.axaml`, foram utilizados os tipos `ListBox` para fazer a lista de conversões, `TextBox` para receber e exibir os valores e `Button` para criar um botão para conversão, além do `TextBloc` para representar um bloco comum de texto. Segue a parte relevante do código:

---

```

<ListBox ItemsSource="{Binding ConversionTypes}" SelectedItem="{Binding SelectedCon
    Grid.Row="0" Grid.Column="1" Margin="10" Height="100">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Name}" />
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
[...]
<TextBox Text="{Binding InputValue}" Grid.Row="1" Grid.Column="1" Margin="10"/>
[...]
<TextBox Text="{Binding OutputValue}" Grid.Row="2"
Grid.Column="1" Margin="10" IsReadOnly="True"/>
[...]
<Button Content="Converter" Command="{Binding ConvertCommand}" Grid.Row="3"
Grid.Column="1" Margin="10" HorizontalAlignment="Right"/>

```

Observa-se principalmente a palavra *Binding* dentro de chaves. Isso ocorre para chamar elementos da divisão ViewModel, que serão conectados automaticamente a elas, como ConvertCommand e Name. Essa funcionalidade do Avalonia é extremamente útil para separar as lógicas de View e backend.

## 2.4 ViewModel

Nesta parte são geradas as propriedades que são utilizadas na View a partir da chamada do comando Binding, além da definição final da função de conversão para utilização pelo programa. A seguir parte do código de definição das propriedades. Perceba que a grande característica é a mudança no set, para além de ele mudar o valor da propriedade *\_nomeDaPropriedade* = *value* chamar o evento OnPropertyChanged

```

public ObservableCollection<ConversionType> ConversionTypes { get; }
private ConversionType _selectedConversionType = null!;
public ConversionType SelectedConversionType
{
    get => _selectedConversionType;
    set
    {
        _selectedConversionType = value;
        OnPropertyChanged();
    }
}

```

```

}

private string _inputValue = string.Empty;
public string InputValue
{
    get => _inputValue;
    set
    {
        _inputValue = value;
        OnPropertyChanged();
    }
}

[...]
```

O evento `OnPropertyChanged` é definido nas linhas abaixo, a partir de convenções padrão para mudanças de propriedade quando utilizado o binding. Ele notifica quando uma propriedade foi alterada, passando seu nome, para a interface atualizá-la.

```

public event PropertyChangedEventHandler? PropertyChanged;

protected void OnPropertyChanged([CallerMemberName] string? propertyName = null)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

A primeira linha define o evento nomeado `PropertyChanged`, que pode ser null pelo ? antes de seu nome, do tipo `PropertyChangedEventHandler`.

A função abaixo define o evento, que tem `[CallerMemberName]`, um atributo mágico que passa automaticamente o nome da propriedade que chamou o evento, caso não for obtido será null, de acordo com o final da linha. Dentro da função se `PropertyChanged` tiver algum inscrito (verificado por ?) o evento é disparado passando o objeto que disparou o evento e um objeto que descreve qual propriedade mudou.

Após a definição das propriedades, com o evento que dispara quando são alteradas, é feito o construtor da classe:

```

public MainWindowViewModel()
{
    ConversionTypes = new ObservableCollection<ConversionType>(ConversionService.Ge
```

```

        ConvertCommand = new RelayCommand(ExecuteConversion);
    }

```

É criada uma `ObservableCollection`, a qual é uma `Collection` notifica quando é alterada. Ela armazena os tipos de conversão criados pelo `ConversionService`. Após isso é criado um comando para a conversão. Ele recebe um objeto `RelayCommand(ExecuteConversion)`.

`RelayCommand` permite que componentes da UI possam executar comandos e verificar que estão disponíveis. Ela recebe o comando que será chamado, se ele está disponível para ser executado, se ele está habilitado ou não segundo os eventos que ele chama e executa o comando passado no construtor. No caso do código, o comando sempre está disponível para ser executado, pois `CanExecute` sempre retorna `true`, e ficará sempre habilitado, pois `CanExecuteChanged` está vazio.

`ExecuteConversion` executa a conversão

```

private void ExecuteConversion()
{
    if (double.TryParse(InputValue, out double input) && SelectedConversionType != null)
    {
        OutputValue = SelectedConversionType.ConversionFormula(input).ToString("F2");
    }
    else
    {
        OutputValue = "Entrada inválida";
    }
}

```

[...]

```

public class RelayCommand : ICommand
{
    private readonly Action _execute;

    public RelayCommand(Action execute)
    {
        _execute = execute;
    }

    public event EventHandler? CanExecuteChanged
    {

```

```
        add { }  
        remove { }  
    }  
  
    public bool CanExecute(object? parameter) => true;  
  
    public void Execute(object? parameter) => _execute();  
}
```

### 3 Resultados

Seleccione o tipo de conversão:

- Celsius para Fahrenheit
- Fahrenheit para Celsius
- Celsius para Kelvin

Valor de entrada:

  

Valor convertido:

  

Converter

Figura 1 – Tela Inicial

Seleccione o tipo de conversão:

- Celsius para Fahrenheit
- Fahrenheit para Celsius
- Celsius para Kelvin

Valor de entrada:

  

Valor convertido:

  

Converter

Figura 2 – Seleção da conversão e digitação do valor

Selecione o tipo de conversão:	<div>Celsius para Fahrenheit</div> <div>Fahrenheit para Celsius</div> <div>Celsius para Kelvin</div>
Valor de entrada:	<input type="text" value="50"/>
Valor convertido:	<input type="text" value="122.00"/>
<div>Converter</div>	

Figura 3 – Resultado da Conversão

Selecione o tipo de conversão:	<div>Pés para Metros</div> <div>Quilômetros para Milhas</div> <div>Milhas para Quilômetros</div>
Valor de entrada:	<input type="text" value="50"/>
Valor convertido:	<input type="text" value="31.07"/>
<div>Converter</div>	

Figura 4 – Quilometros para milhas

Selecione o tipo de conversão:	<div>Litros para Galões</div> <div>Galões para Litros</div> <div>Millilitros para Onças Fluidas</div>
Valor de entrada:	<input type="text" value="50"/>
Valor convertido:	<input type="text" value="189.27"/>
<div>Converter</div>	

Figura 5 – Galões para Litro



## 4 Conclusão

Todos os resultados foram alcançados, e foi adquirido um essencial conhecimento sobre padrões recomendados em C#, como fazer construção de aplicações a partir do modelo MVVM e do uso do Avalonia UI