



Fundamentos de Desenvolvimento com C#

Aula 13: Interfaces

Professor: Rinaldo Ferreira Junior

E-mail: rinaldo.fjunior@prof.infnet.edu.br

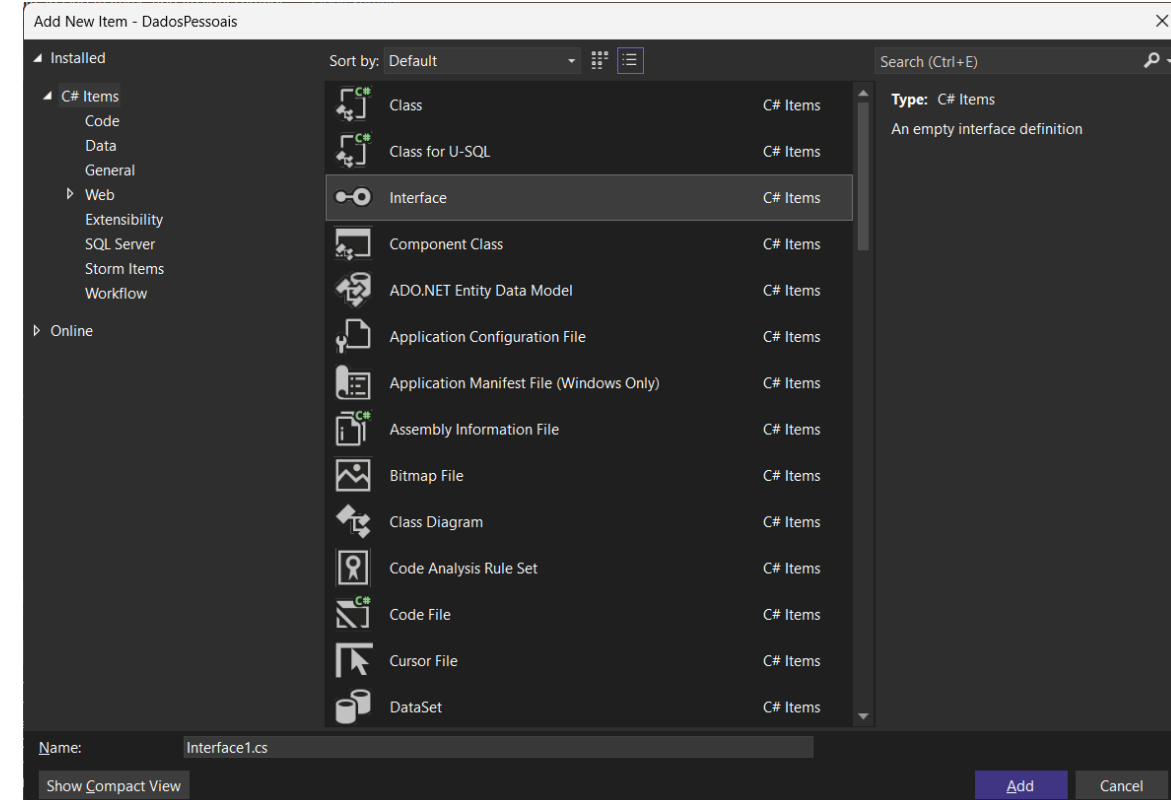
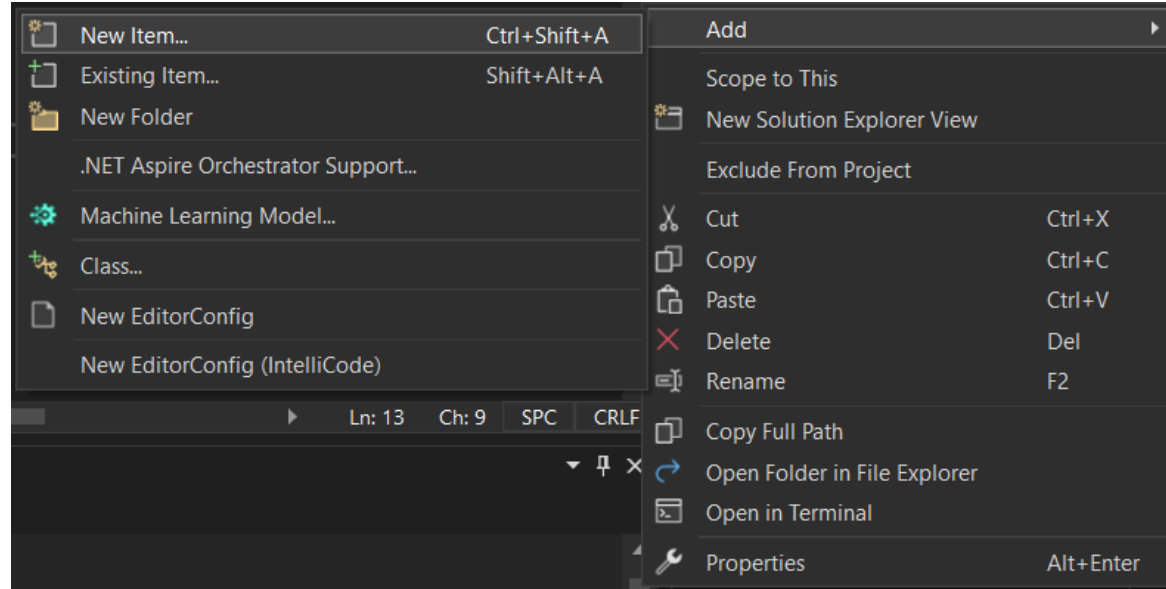


- **Professor:** Rinaldo Ferreira Junior
- **Graduação:** Pós-graduado em Arquitetura de Softwares
- **Atuação:** .Net | C# | SQL | NoSQL | Engenheiro de Software
- **E-mail:** rinaldo.fjunior@prof.infnet.edu.br
- **Linkedin:** <https://www.linkedin.com/in/rinaldo-ferreira-junior-787326a>

- Interfaces

- No âmbito da OO, **Interfaces** são classes completamente abstratas, que definem apenas as assinaturas dos seus membros, sem nenhuma implementação. São chamadas também de contratos.
- A implementação desses membros, é de total responsabilidade das classes concretas que implementarem essa(s) interface(s).
- Por convenção, interfaces são nomeadas com um I maiúsculo como prefixo, por exemplo **IComparable**, **IEnumerable**.
- Uma classe concreta pode implementar múltiplas interfaces, além da herança simples.
- Para implementar uma interface, usa-se também o caractere **:** e separa-se múltiplas interfaces, com vírgulas (,).

- Para criar uma **interface**, clique em **Adicionar – Novo Item** e selecione **interface**.




- Para implementar uma **interface**, use o símbolo **:** após a classe, ou após a classe herdada, se houver.

```
public interface IContaBase
{
    1 reference
    public void Debit(double value);

    1 reference
    public void Debit(int value);

    2 references
    public void Debit(double value, DateTime dataOperacao);

    3 references
    public bool Transfer(double value, int contaDestino, byte digitoDestino);
}
```



```
public class ContaBase : Auditing, IContaBase
```

- Definida a chamada à **interface**, a classe deve implementar os membros definidos na **interface** (contrato).
- A implementação pode ser implícita ou explícita.
- É mais comum o meio implícito. O meio explícito só é necessário se houver métodos iguais em interfaces diferentes.

```
public void Debit(double value, DateTime dataOperacao)
{
    this.Saldo -= value;
    this.DataOperacao = dataOperacao;
}

public override void RegisterEntry(string entry)
{
    throw new NotImplementedException();
}

//Implementação explícita
bool IContaBase.Transfer(double value, int contaDestino, byte digitoDestino)
{
    throw new NotImplementedException();
}

//Implementação implícita
public virtual bool Transfer(double value, int contaDestino, byte digitoDestino)
{
    return true;
}
```

- A partir do C# 8.0, métodos em uma **interface** podem possuir uma implementação padrão.
- Esse recurso permite evoluir uma **interface**, sem quebrar classes que já implementam a **interface**.

```
public interface IContaBase
{
    1 reference
    public void Debit(double value);

    1 reference
    public void Debit(int value);

    2 references
    public void Debit(double value, DateTime dataOperacao);

    0 references
    public void Credit(double value, DateTime dataOperacao) { value = Math.Abs(value); }

    1 reference
    public bool Transfer(double value, int contaDestino, byte digitoDestino);
}
```


- É possível implementar múltiplas **interfaces** em uma classe.

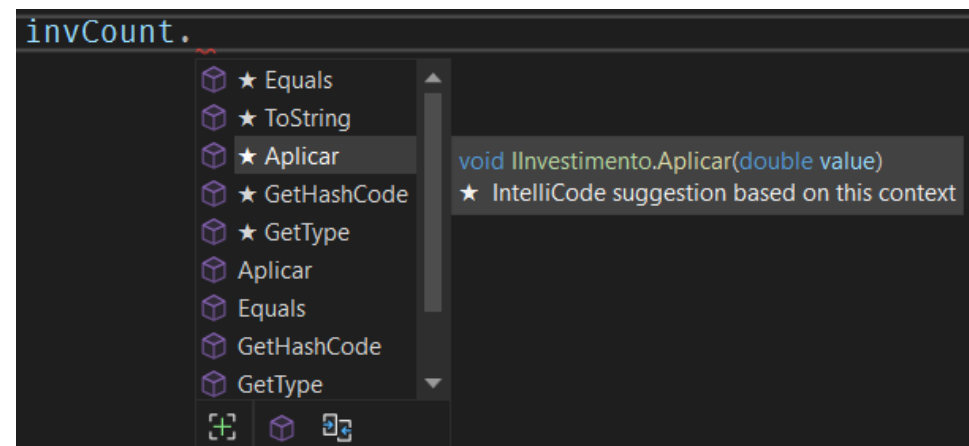
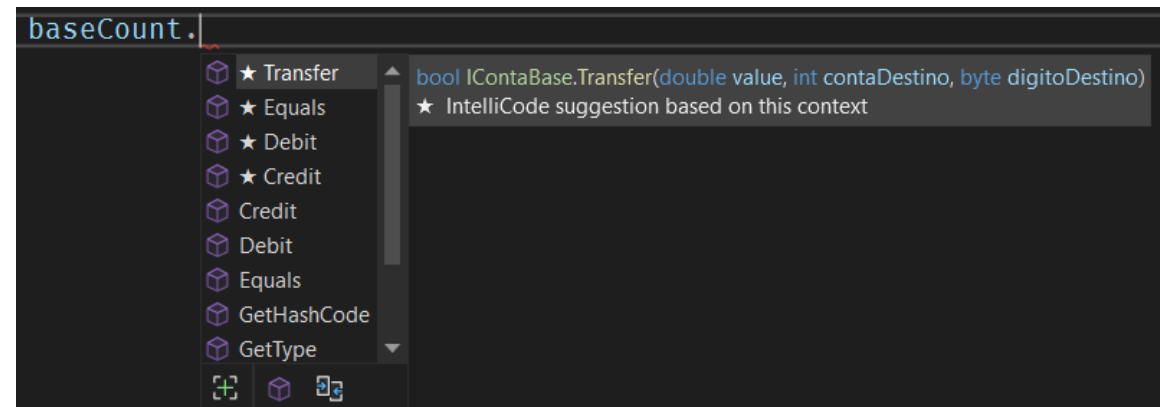
```
public class ContaBancaria : ContaBase, IContaBase, IInvestimento
```

- Com múltiplas interfaces, pode-se criar instâncias com a implementação de cada interface.

```
ContaBancaria conta = new()
{
    Agencia = "18",
    Conta = Utils.CreateAccountNumber(),
    Digito = Utils.CreateAccountDigit(),
    Instituicao = "CEF"
};

ContaBancaria contaAdicional = new()
{
    Agencia = "18",
    Conta = Utils.CreateAccountNumber(),
    Digito = Utils.CreateAccountDigit(),
    Instituicao = "CEF"
};

IContaBase baseCount = conta;
IInvestimento invCount = conta;
```



- O próprio .Net possui diversas **interfaces** que podem ser implementadas em suas classes, para gerar comportamentos específicos.
- Uma dessas interfaces chama-se **IComparable**.
- Essa interface define um método de comparação padrão, que é usado em operações de ordenação. Esse método chama-se **CompareTo()**.
- O método retorna um valor inteiro como resultado da comparação:
 - < 0 -> A instância atual precede o objeto de comparação
 - $= 0$ -> A instância atual está na mesma ordem do objeto de comparação
 - > 0 -> A instância atual sucede o objeto de comparação, na ordenação

- O processo é o mesmo de interfaces personalizadas.

```
public class ContaBancaria : ContaBase, IContaBase, IInvestimento, IComparable<ContaBancaria>
```

```
public int CompareTo(ContaBancaria other)
{
    // Se o saldo for igual então faz a ordenação numérica por conta
    if (this.Saldo == other.Saldo)
    {
        return this.Conta.CompareTo(other.Conta);
    }
    // Ordenação padrão : do maior saldo para o menor
    return other.Saldo.CompareTo(this.Saldo);
}
```

- O método `sort()` da lista, invoca o método `CompareTo()`.

```
static void Main(string[] args)
{
    ContaBancaria conta = new("CEF", 700)
    {
        Agencia = "18",
        Conta = Utils.CreateAccountNumber(),
        Digito = Utils.CreateAccountDigit()
    };

    ContaBancaria contaAdicional = new("CEF", 2300)
    {
        Agencia = "18",
        Conta = Utils.CreateAccountNumber(),
        Digito = Utils.CreateAccountDigit()
    };

    ContaBancaria contaExtra = new("CEF", 200)
    {
        Agencia = "18",
        Conta = Utils.CreateAccountNumber(),
        Digito = Utils.CreateAccountDigit()
    };

    List<ContaBancaria> contas = new();
    contas.Add(conta);
    contas.Add(contaAdicional);
    contas.Add(contaExtra);
    contas.Sort();

    foreach (ContaBancaria item in contas)
    {
        Console.WriteLine($"Conta: {item.Conta}-{item.Digito} | Saldo: {item.Saldo}");
    }

    Console.Read();
}
```