



Fundamentos de Desenvolvimento com C#

Aula 12: Exceções Personalizadas e Sobrescrita de Métodos

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>

- Exceptions Personalizadas
- Sobrescrita de Métodos
- HashSets

- São classes que herdam de `Exception` e são usadas para representar exceções vinculadas às regras de negócio da aplicação.
- Sua classe pode derivar diretamente de `System.Exception` ou de uma das classes `Exception` existentes.
- Recomendações para criação de exceções personalizadas:
 - Crie uma exceção personalizada, apenas se a violação da regra não se encaixar em alguma exceção já existente no CLR.
 - Use a palavra `Exception` como sufixo do nome da classe
 - Crie construtores alternativos
 - Adicione propriedades e métodos relacionados à regra de negócio

- Seguindo as recomendações anteriores, vemos uma exceção personalizada para a operação de Débito.

```
public sealed class InvalidDebitException : Exception
{
    0 references
    public InvalidDebitException()
    {
        ...
    }

    0 references
    public InvalidDebitException(string? message) : base(message)
    {
        ...
    }


    0 references
    public InvalidDebitException(string? message, Exception? innerException) : base(message, innerException)
    {
        ...
    }

    1 reference
    public InvalidDebitException(string? message, double? value) : base(message)
    {
        this.InvalidValue = value;
    }

    2 references
    public double? InvalidValue { get; private set; }
}
```

- O tratamento de uma exceção personalizada é igual ao de qualquer outra exceção.

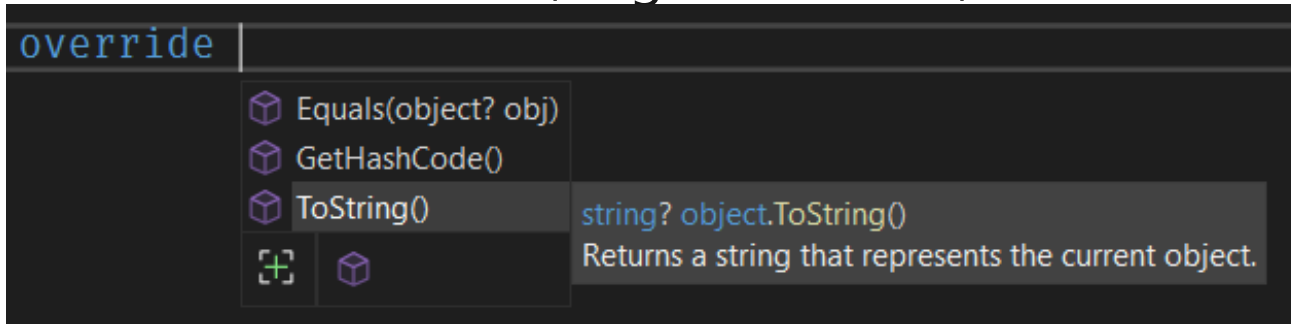
```
public void CheckAndDebit(double saldo, DateTime operacao)
{
    if (saldo < 0)
    {
        throw new InvalidDebitException("O valor informado para débito é inválido", saldo);
    }
    else
    {
        base.Debit(saldo, operacao);
    }
}
```



```
try
{
    conta.CheckAndDebit(-10, DateTime.Now);
}
catch (InvalidDebitException inv)
{
    Console.WriteLine($"{inv.Message} | Valor: {inv.InvalidValue}");
}
catch (Exception)
{
    throw;
}
```

- Classes derivadas podem substituir a implementação de métodos vindos de sua classe base.
- Todas as classes derivam automaticamente de `System.Object` e vários métodos de `System.Object` são substituíveis por padrão. Porém, por padrão, métodos de uma classe não são substituíveis.
- Sobrescrita é desejável quando o método em uma classe base não atende as necessidades da classe especializada, ou, quando diferentes classes especializadas precisam de diferentes implementações.
- Para definir um método como substituível, use a instrução `virtual`. Para efetuar a substituição na classe derivada, use a instrução `override`.


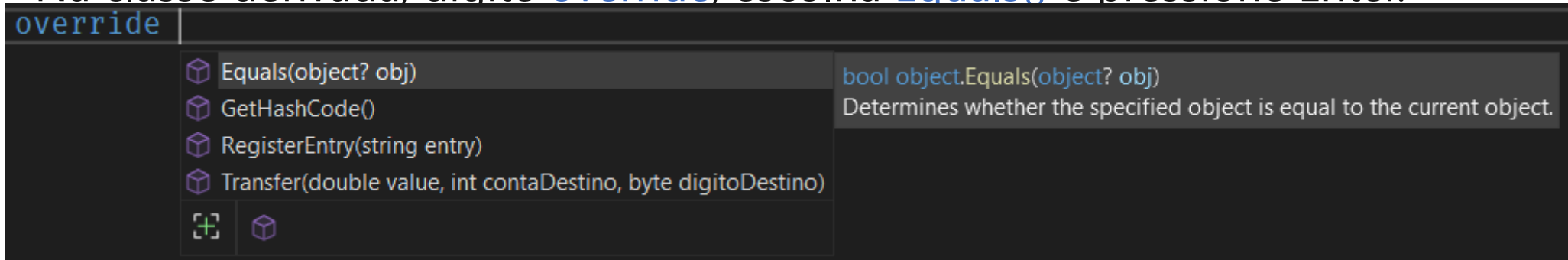
- A classe `System.Object` possui uma implementação padrão para o método `ToString()`. Como esse método é virtual, ele pode ser substituído por implementações especializadas.
- Na classe derivada, digite `override`, escolha `ToString()` e pressione Enter.



```
public override string ToString()
{
    return base.ToString();
}
```

```
public override string ToString()
{
    string resume = $"Banco: {this.Instituicao} | Conta: {base.Conta}-{base.Digito} | Saldo: {base.Saldo}";
    return resume;
}
```


- A classe `System.Object` possui uma implementação padrão para o método `Equals()`. Substituir esse método pode trazer benefícios de performance, ao comparar membros da classe, ao invés de posições de memória.
- Na classe derivada, digite `override`, escolha `Equals()` e pressione Enter.



```
public override bool Equals(object? obj)
{
    if (obj is null)
    {
        return false;
    }

    if (obj is not ContaBancaria)
    {
        return false;
    }

    bool result = (this.Conta == ((ContaBancaria)obj).Conta && this.Digito == ((ContaBancaria)obj).Digito);
    return result;
}
```

- Na comparação direta de classes, `Equals()` compara as referências. Sobrescrevendo, podemos comparar os membros das instâncias.

```
static void Main(string[] args)
{
    ContaBancaria conta = new()
    {
        Agencia = "18",
        Conta = 2710, //Utils.CreateAccountNumber(),
        Digito = 1, //Utils.CreateAccountDigit(),
        Instituicao = "CEF"
    };

    ContaBancaria contaAdicional = new()
    {
        Agencia = "18",
        Conta = 2710, //Utils.CreateAccountNumber(),
        Digito = 1, //Utils.CreateAccountDigit(),
        Instituicao = "CEF"
    };

    bool areSame = conta.Equals(contaAdicional);

    Console.WriteLine($"Contas iguais? {areSame}");
    Console.Read();
}
```

Contas iguais? True

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

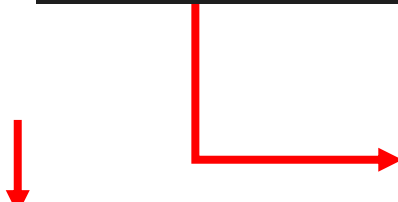
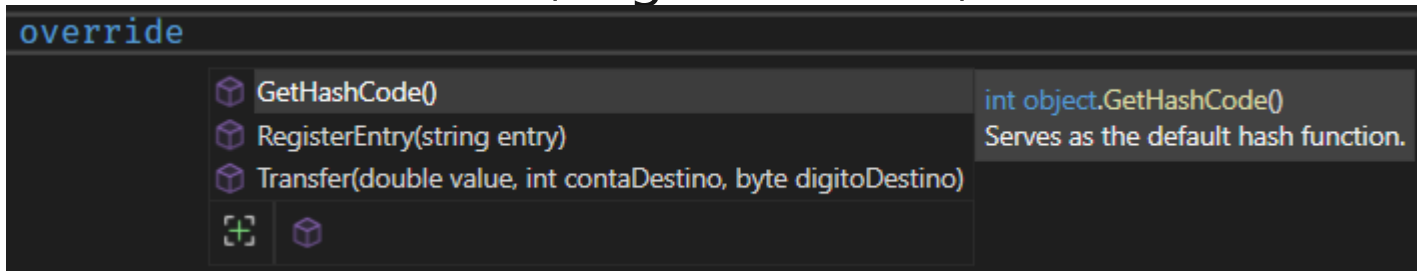
    ContaBancaria contaAdicional = new()
    {
        Agencia = "18",
        Conta = Utils.CreateAccountNumber(),
        Digito = Utils.CreateAccountDigit(),
        Instituicao = "CEF"
    };

    bool areSame = conta.Equals(contaAdicional);

    Console.WriteLine($"Contas iguais? {areSame}");
    Console.Read();
}
```

Contas iguais? False

- O método `GetHashCode()` é invocado automaticamente por classes como `Dicionários` e `HashSets`, que utilizam esse valor para verificar se há elementos iguais no conjunto.
- Na classe derivada, digite `override`, escolha `GetHashCode()` e pressione Enter.



```
public override int GetHashCode()  
{  
    return GetHashCode.Combine(this.Conta, this.Digito);  
}
```

- Observe que o **HashSet** não adiciona elementos com HashCodes iguais.

```
static void Main(string[] args)
{
    ContaBancaria conta = new()
    {
        Agencia = "18",
        Conta = 2710, //Utils.CreateAccountNumber(),
        Digito = 1, //Utils.CreateAccountDigit(),
        Instituicao = "CEF"
    };

    ContaBancaria contaAdicional = new()
    {
        Agencia = "18",
        Conta = 2710, //Utils.CreateAccountNumber(),
        Digito = 1, //Utils.CreateAccountDigit(),
        Instituicao = "CEF"
    };

    bool areSame = conta.Equals(contaAdicional);

    HashSet<ContaBancaria> contasValidas = new();
    contasValidas.Add(conta);
    contasValidas.Add(contaAdicional);

    Console.WriteLine($"Contas iguais? {areSame}");
    Console.Read();
}
```

Locals	
Search (Ctrl+E) 🔍 Search Depth: 3	
Name	Value
args	{string[0]}
conta	{Banco: CEF Conta: 2710-1 Saldo: 0}
contaAdicional	{Banco: CEF Conta: 2710-1 Saldo: 0}
areSame	true
contasValidas	Count = 1

```
static void Main(string[] args)
{
    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"
    };

    bool areSame = conta.Equals(contaAdicional);

    HashSet<ContaBancaria> contasValidas = new();
    contasValidas.Add(conta);
    contasValidas.Add(contaAdicional);

    Console.WriteLine($"Contas iguais? {areSame}");
    Console.Read();
}
```

Locals	
Search (Ctrl+E) 🔍 Search Depth: 3	
Name	Value
args	{string[0]}
conta	{Banco: CEF Conta: 9296-6 Saldo: 0}
contaAdicional	{Banco: CEF Conta: 9910-5 Saldo: 0}
areSame	false
contasValidas	Count = 2

- É um dos tipos de coleção do .Net.
- Internamente implementa um [HashTable](#), contendo pares {key-value}, o que garante performance constante ($O(1)$) em operações básicas como adicionar, remover ou verificar.
- Ao contrário de listas genéricas, não permite elementos duplicados, sendo ideal portanto, para armazenar elementos únicos.
- Automaticamente elimina elementos duplicados, comparando o [HashCode](#) de cada elemento adicionado.
- Não indexa seus elementos, ou seja, não é possível acessar seus elementos por índice. A verificação é feita por meio do método [Contains\(\)](#).
- Não há garantias de que os elementos estarão na ordem de inserção.


- Para tornar seus métodos substituíveis, use a instrução `virtual`.



```
public class ContaBase
{
    0 references
    public ContaBase()
    {
    }

    1 reference
    public ContaBase(double saldo)
    {
    }

    0 references
    public virtual bool Transfer(double value, int contaDestino, byte digitoDestino)
    {
        return true;
    }
}
```

`override`



 RegisterEntry(string entry)	
 Transfer(double value, int contaDestino, byte digitoDestino)	<code>bool ContaBase.Transfer(double value, int contaDestino, byte digitoDestino)</code>
