Aula 3: Tópicos avançados em C#

Professor(a): Virgínia Fernandes Mota
virginia@teiacoltec.org

TECNOLOGIAS DE PROGRAMAÇÃO - SETOR DE INFORMÁTICA

Tópicos avançados em C#

- Ao término desta aula, você será capaz de entender:
  - Exceções
  - Namespaces
  - Classe Object
  - Coleções
  - Conjuntos
  - Manipulação de Strings
  - System I/O
  - LINQ e Lambda
  - Extension Methods
  - Garbage Collector e a CLR

Exceções

```
class Conta
{
    public double Saldo { get; protected set; }
    public bool Saca(double valor)
    {
        if (valor <= this.Saldo)
        {
            this.Saldo -= valor;
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

```
if (conta.Saca(100.0))
{
    MessageBox.Show("Saque efetuado");
}
```

Exceções

- Uma desvantagem dessa abordagem é que se esquecermos de testar o retorno do método Saca, podemos liberar dinheiro pro cliente sem permissão.
- E mesmo invocando o método e tratando o seu retorno de maneira adequada, o que faríamos se fosse necessário sinalizar exatamente qual foi o tipo de erro que aconteceu, como quando o usuário passou um valor negativo como quantidade?
- Uma solução: colocar como boolean. Mas como tratar retornos por motivos diferentes?

Exceções

- No nosso caso, utilizaremos a exceção Exception, indicando que houve um erro na operação de saque:

```
class Conta
{
    public void Saca (double valor)
    {
        if (valor > this.saldo)
        {
            throw new Exception("Valor do saque maior que o saldo")
                ;
        }
        else
        {
            this.saldo -= valor;
        }
    }
}
```

Exceções

- Mas, nós não queremos que o usuário receba tal mensagem na tela.
- Então, não podemos chamar diretamente um método que pode lançar uma exceção.
- Ao invés disso, devemos tentar chamar o método: se não for lançado nenhuma exceção, ok; caso contrário, devemos pegar a exceção e executar um trecho de código referente a exceção.

Exceções

```
private void button2_Click(object sender, EventArgs e)
{
    string textoValorSaque = valorOperacao.Text;
    double valorSaque = Convert.ToDouble(textoValorSaque);
    try
    {
        //cont m o fluxo normal do programa
        contaAtual.Saca(valorSaque);
        MessageBox.Show("Dinheiro Liberado");
    }
    catch (Exception e)
    {
        MessageBox.Show("Saldo insuficiente");
    }
    MostraConta(contaAtual);
}
```

Exceções

- Mas e se o usuário digitar um número negativo? E como criar Exceções da forma que precisamos?

Exceções

- Para criarmos um novo tipo de exceção, precisamos apenas criar uma nova classe que herde de Exception. Vamos criar uma exceção que indica que ocorreu um erro por saldo insuficiente na conta, a SaldoInsuficienteException:

```
1  class SaldoInsuficienteException : Exception{
2      base("Saldo Insuficiente");
3  }
```

Exceções

```
public void Saca (double valor)
{
    if(valor < 0.0)
    {
        throw new ArgumentException(); //Pr pria do C#
    }
    if (valor > this.Saldo)
    {
        throw new SaldoInsuficienteException(); //Classe que
            criamos e herda de Exception
    }
    else
    {
        this.saldo -= valor;
    }
}
```

Exceções

```
private void button2_Click(object sender, EventArgs e)
{
    string textoValorSaque = valorOperacao.Text;
    double valorSaque = Convert.ToDouble(textoValorSaque);
    try
    {
        contaAtual.Saca(valorSaque);
        MessageBox.Show("Dinheiro Liberado");
    }
    catch (SaldoInsuficienteException e)
    {
        MessageBox.Show("Saldo insuficiente");
    }
    catch (ArgumentException e)
    {
        MessageBox.Show("N o     poss vel sacar um valor negativo"
            );
    }
    MostraConta(contaAtual);
}
```

Exceções

- Observações:

```
1  var conta = new Conta();
2  var caixa = new Caixa();
3  conta.Deposita(100.0);
4  conta.Saca(500.0);
5  caixa.Libera(500.0); //Caso a linha 4 lance uma exce  o ,
       esta linha n o ser  executada.
```

- Caso utilizemos o bloco finally (depois do try catch), ele sempre será executado.

Namespaces

- Como organizar nossas classes? O equivalente aos pacotes do Java!
- Using == Import
- Namespace == Package

Classe Object

- No C# todas as classes são filhas da classe Object!
- Método Equals: Compara duas instâncias de objetos, mas temos que reescrevê-lo para fazer a comparação que queremos (equivalente ao toCompare do Java). Lembrando que == compara apenas a referência.
- Método ToString: Transforma a classe em uma String. Para imprimirmos o conteúdo temos que reescrevê-lo (assim como no Java).

Classe Object

```
class Cliente{
    public string Rg { get; set; }

    public override bool Equals(Object objeto)
    {
        Cliente outroCliente = (Cliente) objeto;
        return this.Rg == outroCliente.Rg;
    }
     public override string ToString()
    {
        return "Nome: " + this.Nome + " RG: " + this.Rg;
    }
}
```

```
Cliente c1 = new Cliente("Fulano");
c1.Rg = "12345678-9";
MessageBox.Show(c1.ToString());  //Imprime Nome: Fulano RG:
    12345678-9
Cliente c2 = new Cliente("Fulano");
c2.Rg = "12345678-9";
c1.Equals(c2); //Retorna True
```

Coleções

- Já vimos que manipular Array em Java ou C# não é muito fácil.
- Solução: A Classe List

```
1  var contas = new List<Contas>(); //aqui a declara  o
       impl cita se faz bem  til
2  var c1 = new ContaCorrente("Nomezinho Bonito");
3  contas.Add(c1);
4
5  Conta copiac1 = contas[0];
6
7  contas.Contains(c1); //retorna true
8  contas.Remove(c1); // ou contas.RemoveAt(0);
9  contas.Count; // retorna o n mero de elementos
10
11 foreach (var c in contas){...}  //para iterar
```

Conjuntos

- Como evitar elementos iguais em uma Lista?
- Poderíamos varrer toda a lista em busca do elemento e inseri-lo caso ele não exista: MUITO CUSTOSO.
- Solução: Conjuntos - A Classe HashSet.

```
1  var contas = new HashSet<Contas>(); //Implementa  o de uma
       tabela Hash
```

Conjuntos

- Possui os métodos Add, Remove, Contains, porém não podemos acessar um elemento em determinada posição como contas[0].
- Precisamos iterar para encontrar o elemento.
- Existem ainda outros tipos de conjunto: SortedSet, Dictionary (Parecido com o HashMap do Java).

Manipulação de Strings

```
1 int resposta = 42;
2 string tudo = "A vida, o universo e tudo mais: " + resposta;
3 MessageBox.show(tudo);
```

```
1 int resposta = 42;
2 string texto = "A vida, o universo e tudo mais: "
3 string tudo = string.Format("Fazendo uma concatena  o de {0} com
      {1}", texto, resposta);
4 MessageBox.show(tudo);
```

Manipulação de Strings

```
string dados = "Han Solo, 36, atirou primeiro";
string[] partes = dados.Split(',');

foreach (string parte in partes){
    Console.WriteLine(parte + " \n");
}
/*Ir  imprimir:
Han Solo
36
atirou primeiro */
```

```
string dados = "Eu n o gosto de feij o";
dados = dados.ToUpper().Replace("n o", " ");
MessageBox(dados); //A string passa a ser EU GOSTO DE FEIJ O
```

Vejam mais métodos na library!

System I/O

- Vamos trabalhar um pouco com arquivos texto!

```
1  //Lendo um arquivo
2  if (File.Exists("file.txt")){
3      //Stream    uma classe que l   bytes
4      Stream entrada = File.Open("file.txt", FileMode.Open);//
           Modo Leitura
5      //Vamos ler o arquivo
6      StreamReader leitor = new StreamReader(entrada);
7      //Lendo uma linha do arquivo
8      string linha = leitor.ReadLine();
9      while (linha != null){
10         linha = leitor.ReadLine();
11     }
12     leitor.Close();
13     entrada.Close();
14 }
```

System I/O

```
1  //Escrevendo no arquivo
2  Stream saida = File.Open("file.txt", FileMode.Create); //Modo
       Escrita
3  StreamWriter escritor = new StreamWriter(saida);
4  escritor.Write("Estou escrevendo");
5
6  escritor.Close();
7  saida.Close();
```

System I/O

- Quando não queremos nos preocupar em fechar um recurso que foi aberto (um arquivo, por exemplo), podemos utilizar o using do c#:

```
1 using(declara  o do recurso){
2     c digo que utiliza o recurso que foi aberto
3 }
```

- Assim que o bloco do using termina de executar, o recurso criado é destruído. Podemos utilizar o seguinte código para abrir um arquivo:

```
1 using (Stream entrada = File.OpenRead("entrada.txt", FileMode.
      Open))
2 using (StreamReader leitor = new StreamReader(entrada)){
3     // usa o leitor
4 }
5 // depois do bloco tanto Stream quanto o StreamReader estar o
       fechados.
```

System I/O

- Quando queremos ler uma linha que o usuário digitou no terminal, utilizamos um atributo do tipo TextReader da classe Console chamado In:

```
1  TextReader leitor = Console.In;
2  string linha = leitor.ReadLine();
3  while(linha != null) {
4      // usa o texto da linha atual
5      linha = leitor.ReadLine();
6  }
```

LINQ e Lambda

- Com o que vimos até agora podemos montar programas bem completos em C#.
- Voltando ao nosso exemplo do Banco, uma coisa que podemos fazer é colocar nossas contas em uma List e fazer buscas.

```
1       var contas = List<Conta>();
2       contas.Add (...)
```

- Quando precisamos buscar algo basta

```
1  var contasComMaisDe2000 = List<Conta>();
2  foreach (var c in contas){
3      if (c.Saldo >= 2000){
4          contasComMaisDe2000.Add(c);
5      }
6  }
7  foreach (Conta c in contasComMaisDe2000){
8      MessageBox.show(c.Titular + " tem saldo maior ou igual a
           2000");
9  }
```

- Mas será que existe uma forma mais elegante? E se minha filtragem de dados fosse mais complicado?

LINQ e Lambda

- Language-Integrated Query: LINQ
- Um filtro complicado de listas pode ser criado através de um código mais simples.

```
1  var contas = List<Conta>();
2  ...
3  var filtradas = var c in contas
4                  where c.Saldo >= 2000
5                  select c;
6
7  foreach (Conta c in filtradas){
8      MessageBox.show(c.Titular + " tem saldo maior ou igual a
           2000");
9  }
```

LINQ e Lambda

- Agora quero somar todos os saldos com mais de 2000.
- O LINQ também nos ajuda!!!

```
1  var contas = List<Conta>();
2  ...
3  var filtradas = var c in contas
4                  where c.Saldo >= 2000
5                  select c;
6
7  double somaTotal = filtradas.Sum(c => c.Saldo);
```

- Esse valor **c => c.Saldo** é chamado de Lambda.
- Além do Sum(), temos outros métodos como Min(), Max() e etc.

LINQ e Lambda

1. Considerando a classe conta criada anteriormente, qual seria o código em LINQ para buscar todas as contas em que o nome do titular comece com a letra "G"?
2. Quando estamos fazendo um filtro com o LINQ, podemos usar o método Where disponível em listas e arrays e passar um lambda para ser executado como filtro. Vamos testar essa sintaxe, filtrando as contas mais antigas (número menor que 1000) e com muito saldo disponível (saldo maior que 5000).
3. Qual seria o código em LINQ para contar quantas contas tem saldo > 5000 ?

LINQ e Lambda

- Exercício 1

```
1  var lista = ... //obt m uma lista de contas
2  var filtrados = from c in lista
3                  where c.Titular.StartsWith("G")
4                  select c;
```

- Exercício 2

```
1  var lista = ... //obt m uma lista de contas
2  var filtrados = lista.Where(c => c.Numero < 1000 && c.Saldo >
       5000.0 );
```

- Exercício 3

```
1  var lista = ... //obt m uma lista de contas
2  int quantidade = lista.Count(c => c.Saldo > 5000.0);
```

LINQ e Lambda

- O LINQ também pode ser utilizado para ordenar uma coleção de elementos. Para ordenar uma coleção, podemos utilizar o orderby dentro de uma query do LINQ.

```
1  var numeros = new int[] { 3, 1, 9, 0, 2, 4, 5 };
2  var ordenados = from n in numeros
3                  orderby n
4                  select n;
```

- Além disso, podemos ordenar uma lista de elementos mais complexos, por exemplo uma lista de contas, passando qual será o campo utilizado como critério de ordenação:

```
1  var contas = // pega lista de contas
2  var ordenadasPorSaldo = from c in contas
3                          orderby c.Saldo //podemos usar
                               descending aqui!
4                          select c;
```

LINQ e Lambda

1. Qual código recupera a lista das contas com saldo maior do que 1000 e ordenadas de forma descendente pelo número?

LINQ e Lambda

```
1 var contas = // pega a lista de contas
2 var resultado = from c in contas
3                 where c.Saldo > 1000
4                 orderby c.Numero descending
5                 select c;
```

```
1 var contas = // pega a lista de contas
2 var resultado = contas.Where(c => c.Saldo > 1000)
3                       .OrderByDescending(c => c.Numero);
```

LINQ e Lambda

- Podemos utilizar o LINQ para ordenar uma lista utilizando critérios secundários de ordenação.
- A ordenação por vários critérios é feita utilizando-se o orderby separando cada um dos critérios por vírgula.
- Por exemplo, se quisermos uma lista de contas ordenadas pelo Saldo e pelo Numero, utilizamos o seguinte código:

```
1  from c in contas
2  orderby c.Saldo, c.Numero
3  select c;
```

```
1  var contas = // pega a lista de contas
2  var ordenadas = contas.OrderBy(c => c.Saldo).ThenBy(c => c.
       Numero);
```

Extension Methods

- Estendendo comportamentos através de métodos extras! Mas eles nunca sobrescrevem o comportamento a classe original!!
- Vamos supor que queremos criar um método para transformar palavras no plural:

```
1  static class StringUtil{
2    public static string Pluralize(string texto)  {
3      if(texto.EndsWith("s")){
4        return texto;
5      }
6      else{
7        return texto + "s";
8      }
9    }
10 }
```

- Para utiliza-lo:

```
1  string contas = StringUtil.Pluralize("conta");
```

- Mas e se eu quiser torna-lo mais elegante?

Extension Methods

- E se eu quisesse fazer algo como

```
1  string texto = "banco";
2  string plural = texto.Pluralize();
```

- Nesse caso estou tentando criar um novo método para string... Como fazer???
- O C# permite que criemos métodos de extensão a classes que já existem através do uso da palavra using.
- Para isso devemos colocar nossa classe estática dentro de um namespace e adicionar a palavra this ao primeiro parâmetro!

Extension Methods

```
namespace MinhasExtensoes {
  public static class StringExtensions{
    public static string Pluralize(this string texto){
      if(texto.EndsWith("s")){
        return texto;
      }
      else{
        return texto + "s";
      }
    }
  }
}
```

- Assim podemos fazer

```
1  using MinhasExtensoes;
2
3  string texto = "banco";
4  string plural = texto.Pluralize();
```

Extension Methods

- É importante lembrar que o método só pode ser acessado caso não exista ainda um outro método com o mesmo nome e tipos de parâmetros na classe.
- Isto é, não seria possível estender a classe string com um novo método ToString() pois ele já existe. Somente podemos adicionar novos comportamentos.
- Outro ponto importante é que o this só funciona no primeiro argumento.

Garbage Collector e a CLR

- Problemas?
- Precisamos sempre recompilar o código para o SO que iremos utilizar ou máquina.

Garbage Collector e a CLR

- Interpretador == Command Language Runtime == CLR $\rightarrow$ Máquina Virtual
- Intermediário == Intermediate Language == IL

Garbage Collector e a CLR

- Classes escritas em diferentes linguagens mas compiladas para IL podem se comunicar.
- A CLR otimiza a execução do código.
- **new**: Cria um novo objeto gerenciado pela CLR na memória $\rightarrow$ Garbage Collector

Na próxima aula...

Programação Concorrente