

Disciplina: POO - Programação Orientada a Objetos

TRATAMENTO DE EXCEÇÃO

Prof^a. Dr^a. Giovana Angélica Ros Miola
giovana.miola@fatec.sp.gov.br



Exceção - Exception

- Uma exceção representa um comportamento inesperado interrompendo o fluxo normal das instruções durante a execução de um programa.
- Geralmente quando o sistema captura alguma exceção o fluxo do código fica interrompido.
- Para conseguir capturar uma **exceção**, é preciso fazer antes o tratamento.

Exceção - Exception

- Quando uma exceção ocorre, elas são chamadas de ***throw***.
- O ***throw*** atual é um objeto que é derivado da classe ***System.Exception***.
- Em C# erros de execução são definidos por classes que derivam direta ou indiretamente da classe ***System.Exception***
- Erros conhecidos: divisão por zero ou tentar escrever em um arquivo somente leitura

Exceção - Exception

- O uso dos tratamentos é importante nos sistemas, porque auxilia em falhas como: comunicação, leitura e escrita de arquivos, entrada de dados inválidos, acesso a elementos fora de índice, falta de memória, impressora sem papel, acesso a banco de dados, entre outros.
- Nestes trechos de códigos, deve-se colocar o tratamento de exceção para que o programa não seja abortado e ou que não passe uma informação técnica fora do alcance do usuário final.

Exemplo com System.IO;

- Quando se faz necessário um tratamento de exceção, pode-se tratar vários erros em apenas uma estrutura.
- Os tratamentos mais ambíguos devem ficar entre os últimos e os mais específicos devem ficar entre os primeiros.
- Veja o exemplo, usando o **System.IO**;

```
using System;
using System.IO;
namespace ExcecaoArquivo
{
    class Exemplo
    {
        public static void Main()
        {
            try
            {
                // Isto causará uma exceção
                File.OpenRead(@"\teste\ArquivoLeitura.txt");
            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine();
                Console.WriteLine("Erro pelo FileNotFoundException");
                Console.WriteLine("-----");
                Console.WriteLine(e.ToString());
                Console.WriteLine();
            }
            catch (DirectoryNotFoundException e)
            {
                Console.WriteLine();
                Console.WriteLine("Erro pelo DirectoryNotFoundException");
                Console.WriteLine("-----");
                Console.WriteLine(e.ToString());
                Console.WriteLine();
            }
            catch (Exception e)
            {
                Console.WriteLine();
                Console.WriteLine("Erro pelo Exception");
                Console.WriteLine("-----");
                Console.WriteLine(e.ToString());
                Console.WriteLine();
            }
        }
    }
}
```

Supondo que você está tentando acessar um arquivo remoto, no catch pode ter uma comando para enviar um e-mail pedindo solução para o problema.

Exemplo com System.IO.File

- A classe **System.Exception** contém alguns métodos e propriedades para obter informações sobre o erro que foi gerado.
- Por exemplo, o **Message** é uma propriedade que contém um resumo do erro gerado; **StackTrace** retorna informações do erro ocorrido; o método **ToString()** retorna uma descrição completa do erro ocorrido.
- Existem vários tipos de Exceções. Por exemplo, na classe **System.IO.File** o método **OpenRead()**, tem as seguintes exceções:
 - *SecurityException;*
 - *ArgumentException;*
 - *ArgumentNullException;*
 - *PathTooLongException;*
 - *DirectoryNotFoundException;*
 - *UnauthorizedAccessException;*
 - *FileNotFoundException;*
 - *NotSupportedException.*

Retorno do método para controlar erros

- Exemplo:

- Se sacar um valor superior ao saldo do cliente, o método não permitirá o saque
- Quem chamou o método não saberá se o saque ocorreu ou não

Retorno do método para controlar erros

- Como avisar quem invocou o método, que o saque foi feito com sucesso ou não?
 - Possível solução: Alterar o método Sacar para retornar um valor booleano, indicando se o saque foi ou não efetuado

Retorno do método para controlar erros

```
class Conta
```

```
{
```

```
//....
```

```
public bool Sacar (double valor)
```

```
{
```

```
    if (valor <= this.Saldo)
```

```
        Saldo -= valor
```

```
        return true;
```

```
    return false;
```

```
}
```

```
}
```

```
// no método Main().....
```

```
Conta conta = new Conta();
```

```
if (conta.Sacar(100.0))
```

```
{
```

```
    Console.WriteLine("Saque efetuado");
```

```
}
```

Retorno do método para controlar erros

- É necessário saber se o saque foi efetuado ou não, antes de liberar o dinheiro para o cliente.
- De acordo com esse desenvolvimento NÃO SE PODE ESQUECER de testar o retorno do método Sacar, se não, será liberado o dinheiro para o cliente sem permissão.

Retorno do método para controlar erros

- E mesmo invocando o método e tratando o seu retorno de maneira adequada, o que seria necessário para sinalizar exatamente?
- Qual foi o tipo de erro que aconteceu, como quando o usuário passou um valor negativo como quantidade?
 - Possível solução: Alterar o retorno de bool para número inteiro e retornar o código do erro que ocorreu.

Retorno do método para controlar erros

- Mas isso seria uma má prática, pois qual seria o valor devolvido?
- Como seria tratado?
- Além de não obrigar o programador a tratar esse retorno, o que pode levar o programa a continuar executando ainda em um estado inconsistente.

Retorno do método para controlar erros

- Um outro problema aconteceria se o método já retornasse algum valor.
- Desse jeito, não daria para alterar o retorno para indicar se o saque foi realizado ou não.

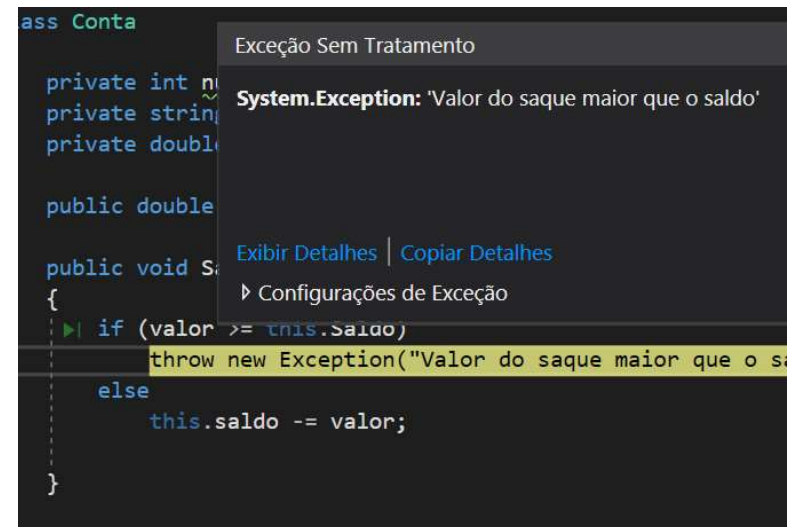
Controlando erros com exceção

- Para evitar esses problemas, será usada a **exception** para tratar essas exceções.
- Em vez de, retornar um valor dizendo se uma operação foi bem sucedida, é lançada uma exceção à regra padrão, dizendo que algo de errado aconteceu.
- Neste caso, utiliza a exceção **Exception**, indicando que houve um erro na operação de saque, como o exemplo a seguir:

Controlando erros com exceção

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

A exceção lançada pelo
throw tem que ser
tratada no bloco catch



The screenshot shows a code editor with a C# class named 'Conta'. The 'Sacar' method is highlighted, and a tooltip is displayed over the 'throw new Exception' line. The tooltip title is 'Exceção Sem Tratamento' and the message is 'System.Exception: 'Valor do saque maior que o saldo''. Below the message are links for 'Exibir Detalhes' and 'Copiar Detalhes', and a section for 'Configurações de Exceção'.

```
class Conta
{
    private int n...
    private string...
    private double...

    public double...

    public void S...
    {
        if (valor >= this.Saldo)
            throw new Exception("Valor do saque maior que o s...");
        else
            this.saldo -= valor;
    }
}
```

Controlando erros com exceção

- Mas, não seria interessante o usuário receber tal mensagem na tela?
- Então, não se pode chamar diretamente um método que pode lançar uma exceção.
- Ao invés disso, é interessante no local oportuno, usar os comandos de tratamento de exceção, ou também se quiser, criar um método para chamar o Sacar()

Controlando erros com exceção

- Caso contrário, deve-se pegar a exceção e executar um trecho de código referente à exceção.
- Para tentar executar um trecho de código que pode lançar uma exceção, deve-se colocá-lo dentro de um bloco **try**.
- Neste caso, a chamada do método Sacar(), será colocado dentro do bloco que trata a exceção

Controlando erros com exceção

```
... Main()
{
    Conta c = new Conta();
    c.Saldo = 100;
    try
    {
        c.Sacar(101);
        Console.WriteLine("Dinheiro Liberado");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Saldo insuficiente");
    }
}
```

- Nesse exemplo, caso o método Sacar lance uma exceção, o bloco catch será executado mostrando a mensagem “Saldo Insuficiente”.
- No caso de uma exceção, a mensagem “Dinheiro Liberado” não é exibida.
- O bloco try deve conter toda a lógica de negócio que será executada em uma situação normal, quando não ocorrem casos excepcionais.
- A primeira situação é se preocupar com a lógica de negócios e depois com os erros que aconteceram.

Comparação entre try.. catch e if

- São diferentes entre si
- **if** é utilizado para fazer um fluxo condicional
- **try .. catch** é utilizado para pegar exceções e dar a essas o tratamento adequado
- Exceções só devem ser usadas em situações excepcionais, pois são consideradas lentas

Exceção - Exception

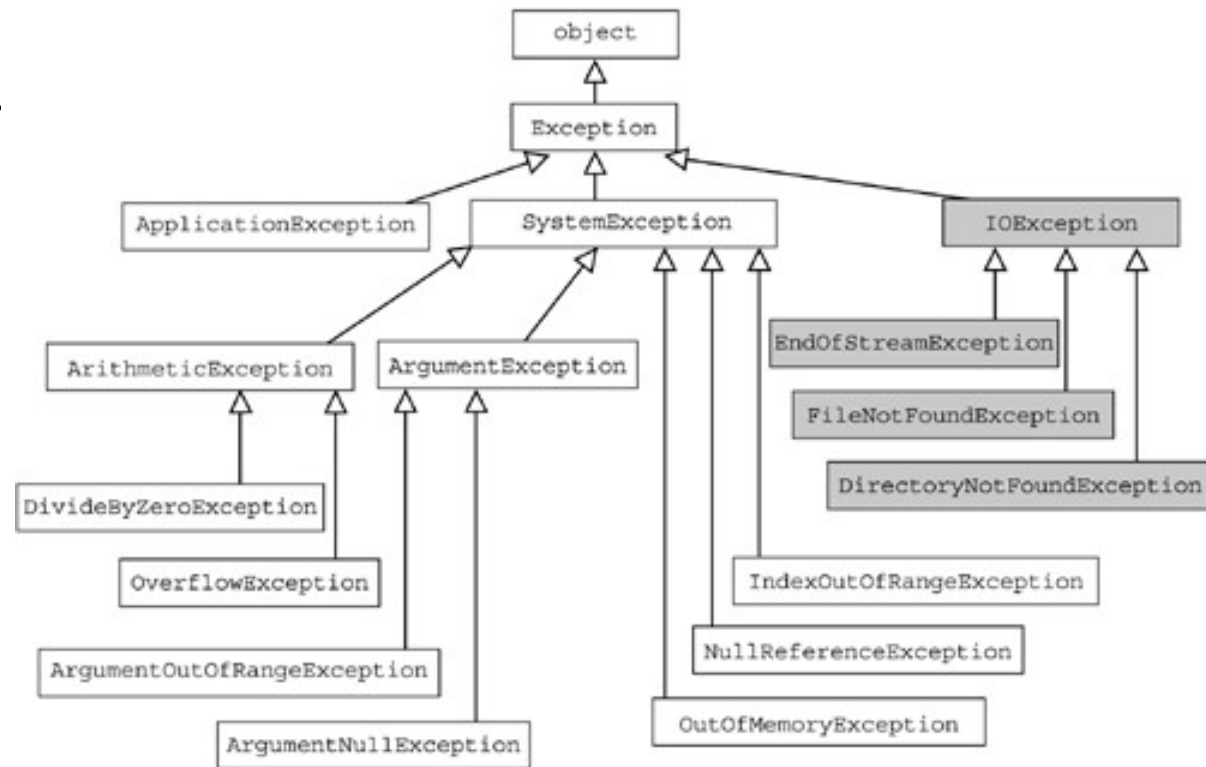
- As instruções utilizadas pelo C# para tratamento de exceções, são:

Comando	Descrição
try	Bloco onde pode ocorrer o erro
catch	Bloco onde será tratado o erro
finally	Essa instrução é sempre executada, útil quando é necessário liberar um recurso independentemente se ocorrer erro ou não
throw	Comando responsável para gerar exceções específicas

- O **try** e o **catch** são obrigatórios, porém, não é necessário ter mais de um catch para cada tratamento de exceção

Hierarquia de classes de exceção

- Existem diversos tipos de exceção que podem ocorrer num programa



Principais classes de exceção

- Principais classes derivadas de System.Exception

Exception	Descrição
DivideByZeroException	Erro gerado quando dividimos números inteiros por zero
IndexOutOfRangeException	Erro gerado quando acessamos posições inexistentes de um array
NullReferenceException	Erro gerado quando utilizamos referências nulas
InvalidCastException	Erro gerado quando realizamos um casting incompatível

Lançando erros - throw

- Para lançar um erro, deve-se **criar um objeto** de qualquer classe que deriva de Exception para representar o erro que foi identificado
- Depois de criar uma exceção pode-se “lançar” a referência dela utilizando o comando **throw**
- Observe o exemplo utilizando a classe System.ArgumentException que deriva indiretamente da classe System.Exception

```
if ( valor < 0)
{
    System.ArgumentException erro = new System.ArgumentException ();
    throw erro ;
}
```

Lançando erros - throw

- O comando **throw** invoca a Exceção gerada

```
int valor1 = 100, valor2 = 0, total = 0;
try //Bloco onde poderá ocorrer algum erro
{
    total = valor1 / valor2;
}
catch (Exception ex) //Classe responsável pelo tipo de erro
{
    throw ex;
}
```


Capturando erros – try .. catch

- As instruções do **try** são sempre executadas, e do **catch** somente serão executadas se a instrução anterior gerar uma exceção
- No exemplo anterior é forçada a exceção, dividindo um valor por zero.
- A classe genérica “Exception” proporciona visualizar o tipo de erro utilizando a propriedade “ex.Message”

Capturando erros – try .. catch

```
int valor1 = 100, valor2 = 0, total = 0;
try //Bloco onde poderá ocorrer algum erro
{
    total = valor1 / valor2;
}
catch (Exception ex) //Classe responsável pelo tipo de erro
{
    Console.WriteLine("Ocorreu uma exceção na rotina acima! "+
                      ex.Message);
}
```

Capturando erros – try .. catch

```
Conta c = new Conta ();
```

```
try
```

```
{
```

```
    c.Deposita (100) ;
```

```
}
```

```
catch ( System.ArgumentException e)
```

```
{
```

```
    System.Console.WriteLine (" Houve um erro ao depositar ");
```

```
}
```

Capturando erros – try .. catch

- Pode-se encadear vários blocos catch para capturar exceptions de classes diferentes

```
Conta c = new Conta ();
```

```
try
```

```
{
```

```
    c.Deposita (100) ;
```

```
}
```

```
catch ( System.ArgumentException e)
```

```
{
```

```
    System.Console.WriteLine (" Houve um System.ArgumentException ao depositar ");
```

```
}
```

```
catch ( System.IO.FileNotFoundException e)
```

```
{
```

```
    System.Console.WriteLine (" Houve um FileNotFoundException ao depositar ");
```

```
}
```

finally

- Se um erro acontecer no bloco **try** ele é abortado. Consequentemente, nem sempre todas as linhas do bloco **try** serão executadas.
- Somente um bloco **catch** é executado quando ocorre um erro.
- Em alguns casos, é necessário executar um trecho de código independentemente se houver erros ou não. Para isso pode-se, utilizar o bloco **finally**.

finally

```
try
{
    // código
}
catch ( System.DivideByZeroException e)
{
    System.Console.WriteLine (" Tratamento de divisão por zero ");
}
catch ( System.NullReferenceException e)
{
    System.Console.WriteLine (" Tratamento de referência nula ");
}
finally
{
    // código que deve ser sempre executado, independente do código ter sido o do try ou do catch
}
```

finally

```
int valor1 = 100, valor2 = 0, total = 0;
try //Bloco onde poderá ocorrer algum erro
{
    total = valor1 / valor2;
}
catch (Exception ex) //Classe responsável pelo tipo de erro
{
    Console.WriteLine("Ocorreu uma exceção na rotina acima! ", ex.Message);
}
finally //Bloco que sempre será executado, ocorrendo ou não erros no bloco try
{
    Console.WriteLine("Mesmo não ocorrendo uma exceção acima esta mensagem será exibida!");
}
```

Exemplo: Você calculará algo que veio os valores via banco de dados, ou seja sua conexão esta aberta, no finally você pode ter o comando de fechamento de conexão, independente do código do try ou catch

Classe Exception

- Algumas propriedades

Propriedades	Descrição
HelpLink	Possui o caminho apontando para um arquivo contendo informações de ajuda sobre a exceção que ocorreu
InnerException	Referência uma exceção interna
Message	Contém a mensagem de erro
Source	Retorna informações do sistema que gerou o erro
StackTrace	Esta propriedade contém o rastreamento de pilha. Ela pode ser usada para determinar a localização da ocorrência do erro
TargetSite	Informações sobre o método que lançou a exceção

Uso de algumas propriedades de exceção

```
int valor1 = 100, valor2 = 0, total = 0;
try //Bloco onde poderá ocorrer algum erro
{
    total = valor1 / valor2;
}
catch (Exception ex) //Classe responsável pelo tipo de erro
{
    Console.WriteLine( "Ocorreu uma exceção na rotina acima! \n" +
        "Tipo Erro: " + ex.Message + "\n" + "Pilha de execução: " + ex.StackTrace + "\n" +
        "Informações do Sistema: "+ex.Source + "\n" + "Método: "+ex.TargetSite + "\n");
}
```

Dicas para uso do tratamento de exceção

- Tratamento de exceções — resolver exceções que poderiam ocorrer para que o programa continue ou termine elegantemente.
- O tratamento de exceções permite que os programadores criem programas mais robustos e tolerantes a falhas.
- O tratamento de exceção ajuda a aprimorar a tolerância a falhas de um programa.

Dicas para uso do tratamento de exceção

- Evite utilizar o tratamento de exceções como uma forma alternativa de fluxo de controle.
- Essas “exceções adicionais” podem interferir nas verdadeiras exceções do tipo erro.
- Evite colocar código que possa lançar/jogar (throw) uma exceção em um bloco finally. Se esse código for necessário, inclua o código em um try...catch dentro do bloco finally.

Dicas para uso do tratamento de exceção

- Se os problemas potenciais ocorrem raramente, mesclar o programa e a lógica do tratamento de erro pode degradar o desempenho de um programa, porque o programa deve **realizar testes** (potencialmente frequentes) para determinar se a tarefa foi executada corretamente e se a próxima tarefa pode ser realizada.
- Com o tratamento de exceções, um programa pode continuar executando (em vez de encerrar) depois de lidar com um problema. Isso ajuda a assegurar o tipo de aplicativos robustos que colaboram para o que é chamado de computação de missão crítica ou computação de negócios críticos.

Criar exceções personalizadas

- Para criar uma exceção sua, inclua ao final do nome a palavra Exception
- Use pelo menos três dos construtores comuns ao criar suas próprias classes de exceção
- Incluir três construtores em classes de exceção personalizada
 - Exception(), que usa valores padrão
 - Exception(String), que aceita uma mensagem
 - Exception(String, Exception), que aceita uma mensagem e a exceção interna.

```
class MinhaClassePersonalizadaException : Exception
{
    //....código necessário
    public MinhaClassePersonalizadaException(string mensagem):base(mensagem)
    {
        //Construtor que chama o construtor da classe genérica
    }
}
```