

Aula 1: Conceitos e importância do tratamento de exceções

Tratamento de Erro em C# com
Exceptions

Objetivos

1. Conceituando exceções e seus tipos
2. Importância do tratamento de erros

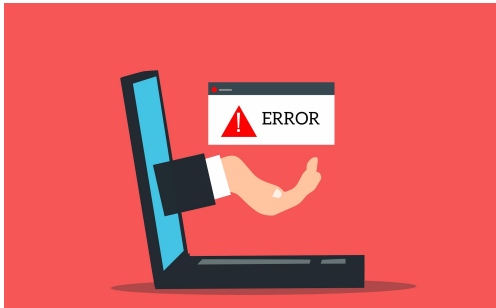
Etapa 1

Exceções: conceito e tipos

Tratamento de Erro em C# com
Exceptions

O que é uma exceção?

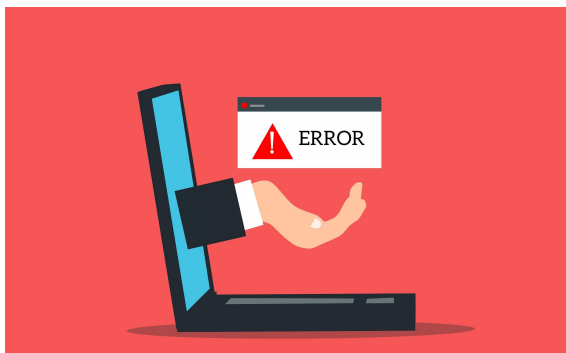
- ★ Qualquer condição de erro ou comportamento inesperado por uma programa em execução.
 - falha de codificação
 - falta de recursos disponíveis
 - condições inesperadas pelo runtime



Tipos de erros

★ Tipos de erros possíveis

- Erros de sintaxe
- Erros em tempo de execução
- Erros lógicos





Erro lógico

- ★ Exceção provocada por falha lógica do desenvolvedor.
- ★ Deve ser tratado a partir da correção do código falho.

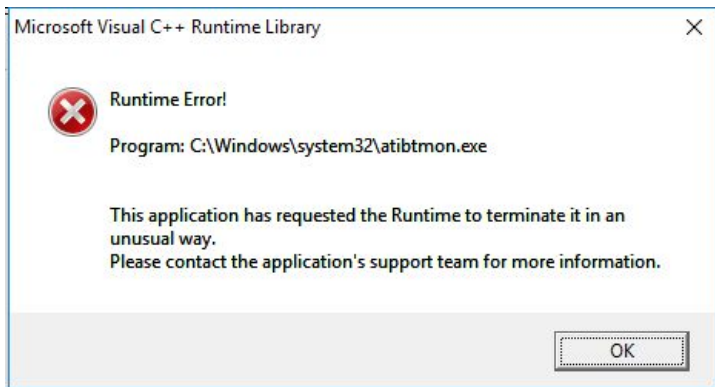
```
public static void Main()
{
    Person p1 = new Person();
    p1.Name = "John";
    Person p2 = null;

    // The following throws a NullReferenceException.
    Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
}
```



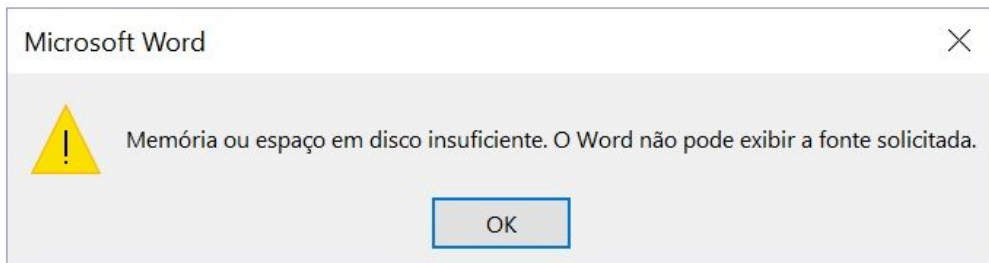
Erros de tempo de execução

- ★ Erros em tempo de execução que não estão necessariamente relacionados à código mal escrito.
 - comum em caso de leitura/escrita de arquivos



Falha de sistema

- ★ Erro de tempo de execução que não pode ser tratado programaticamente de maneira significativa.
→ falta de recursos





DIGITAL
INNOVATION
ONE

Etapa 2

Importância do tratamento de erros

Tratamento de Erro em C# com
Exceptions



Por que devemos tratar os erros?

- ★ Evitar parada súbita do sistema
- ★ Mensagens amigáveis para usuário final
- ★ Melhor comunicação com desenvolvedores para tratar rapidamente o problema

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
com.potix.zk.ui.UiException: Recursive import: /test/import.zul  
    com.potix.zk.ui.metainfo.Parser.parse(Parser.java:200)  
    com.potix.zk.ui.metainfo.Parser.parse(Parser.java:90)  
    com.potix.zk.ui.metainfo.PageDefinitions$MyLoader.parse(PageDefinitions.java:186)  
    com.potix.web.util.resource.ResourceLoader.load(ResourceLoader.java:94)  
    com.potix.util.resource.ResourceCache$Info.load(ResourceCache.java:223)  
    com.potix.util.resource.ResourceCache$Info.<init>(ResourceCache.java:197)  
    com.potix.util.resource.ResourceCache.get(ResourceCache.java:136)
```



Por que devemos tratar os erros?

- ★ Evitar parada súbita do sistema
- ★ Mensagens amigáveis para usuário final
- ★ Melhor comunicação com desenvolvedores para tratar rapidamente o problema



Vamos entender na prática?

★ Usando Visual Code

*Vamos ver alguns
exemplos?*

★ Usando Visual Studio



DIGITAL
INNOVATION
ONE

Aula 2: Entendendo a classe `System.Exception`

Tratamento de Erro em C# com
Exceptions



Objetivos

1. Entender a hierarquia de classe de exceções a partir da classe base `System.Exception` e exceções mais comuns
2. Conhecer propriedades e métodos úteis
3. Aprender instruções associadas ao tratamento de Exceções

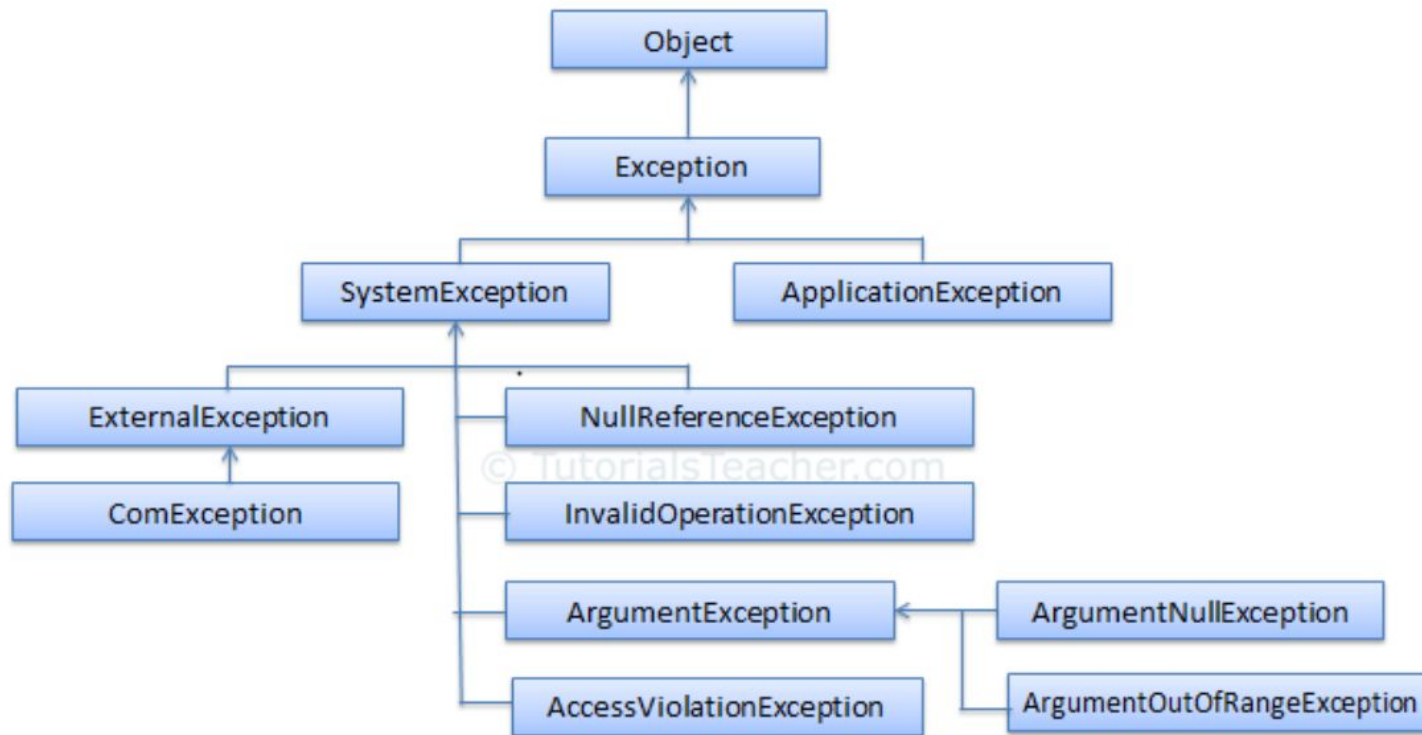
Etapa 1

Hierarquia de classes exception

Tratamento de Erro em C# com
Exceptions



Hierarquia de classes





A classe *Exception*

- Toda exceção em .NET herda da classe *System.Exception*
- Algumas exceções comuns([Exception Classe \(System\) | Microsoft Docs](#)):

IndexOutOfRangeException	DivideByZeroException
NullReferenceException	FileNotFoundException
InvalidOperationException	StackOverflowException
Argument Exception	OutOfMemoryException
ArgumentOutOfRangeException	KeyNotFoundException

Etapa 2

Propriedades e métodos úteis da classe Exception

Tratamento de Erro em C# com
Exceptions



A classe *Exception*: *Propriedades*

- Propriedades importantes herdadas:
 - ◆ Message: descrição legível para humanos com a causa da exceção
 - ◆ InnerException: obtém conjunto de exceções superiores ou exceção que levou à exceção atual.
 - ◆ StackTrace: rastreamento do caminho até chegar ao erro



A classe *Exception*: *Propriedades*

- Propriedades importantes herdadas:
 - ◆ Source: relacionada à aplicação ou objeto que causou o erro
 - ◆ TargetSite: relacionada ao método que lançou a exceção atual



A classe *Exception*: métodos

1.

```
public virtual Exception GetBaseException ();
```

Dada uma cadeia de exceções, somente uma delas pode ser a causa raiz para todas as outras, portanto é a *'exceção base'*.



A classe *Exception*: métodos

2.

```
public virtual void GetObjectData  
(System.Runtime.Serialization.SerializationInfo info,  
System.Runtime.Serialization.StreamingContext context);
```

- *Configuração de informações relacionadas à exceção lançada.*
- ◆ *info: guarda objeto de dados serializados*
 - ◆ *context: contém informação de contexto sobre origem ou destino dos dados a serem transmitidos*



A classe *Exception*: métodos

3.

```
public Type GetType ();
```

→ *Retorna tipo da instância atual em tempo de execução.*



A classe *Exception*: métodos

4.

```
public override string ToString ();
```

Vamos
exemplificar?

→ *Retorna representação da atual exceção em forma de string*



DIGITAL
INNOVATION
ONE

Etapa 3

Comandos para lançar e tratar exceções

Tratamento de Erro em C# com
Exceptions

Lidando com exceções

★ Comando *try*

Provê mecanismo para capturar exceções que ocorrem durante a execução de certo bloco de código.



E após a captura?



Lidando com exceções

★ Comando *catch*

Após a captura, o sistema procura pelo comando `catch` mais próximo que pode lidar com a exceção.

```
try
{
}
catch (Exception exception)
{
    Console.WriteLine(exception.Message);
}
```

Pode-se usar
mais de um
bloco *catch*!

Lidando com exceções

★ Comando *finally*

Bloco útil para liberação de recursos, pois sempre é executado, independente da captura e tratamento da exceção.

```
try
{
    file.ReadBlock(buffer, index, buffer.Length);
}
catch (System.IO.IOException e)
{
    Console.WriteLine("Error reading from {0}. Message = {0}", e.Message);
}
finally
{
    if (file != null)
    {
        file.Close();
    }
}
```

Lidando com exceções

★ Comando *throw*

Lança uma exceção em código explicitamente.

```
throw new DivideByZeroException();
```

- Utilizar o comando em um contexto de exceção já capturada, faz o “relançamento” da exceção dentro do catch, provendo assim mais informação para depuração.

Lidando com exceções

★ Comando *when*

Trata exceções de acordo com requerimentos específicos que você define para dada exceção.

→ Útil quando uma exceção pode ser tratada igualmente para múltiplos erros sob determinadas condições.



DIGITAL
INNOVATION
ONE

Aula 3: Customizando exceções

Tratamento de Erro em C# com
Exceptions

Objetivos

1. Compreender motivação e como implementar classes customizadas a partir de exemplo prático

Customizando exceções

- ★ Apesar da hierarquia de classes existentes com base na classe Exception, é possível criar sua própria classe de exceção de acordo com a necessidade.
- ★ Motivações:
 - Quando uma exceção reflete um erro específico que não foi mapeado por uma classe de exceção existente
 - Quando a exceção necessita de um tratamento diferenciado

Criando exceções customizadas

★ Procedimento:

1. Definir uma classe que herda de Exception
2. Definir construtores da classe
3. Se necessário, sobrescreva membros cujo comportamento queira modificar
4. Definir se a exceção será serializável



DIGITAL
INNOVATION
ONE

Aula 4: Boas práticas no tratamento de exceções

Tratamento de Erro em C# com
Exceptions

Objetivos

1. Aprender boas práticas no tratamento de exceções



Melhores práticas para exceções

1. Use ***try/catch/finally*** em trechos de códigos que podem potencialmente gerar exceções e que de fato seu próprio código também consegue tratar.
2. Nos blocos ***catch*** , sempre ordene os tratamentos das exceções das classes mais específicas para mais genéricas.
3. Faça limpeza automática de recursos alocados com ***using*** . Caso o objeto não implemente *IDisposable* utilize ***finally***.



Melhores práticas para exceções

4. Caso exista uma condição com grandes chances de erro, verifique a viabilidade de checar a condição antes de somente tratar a exceção.

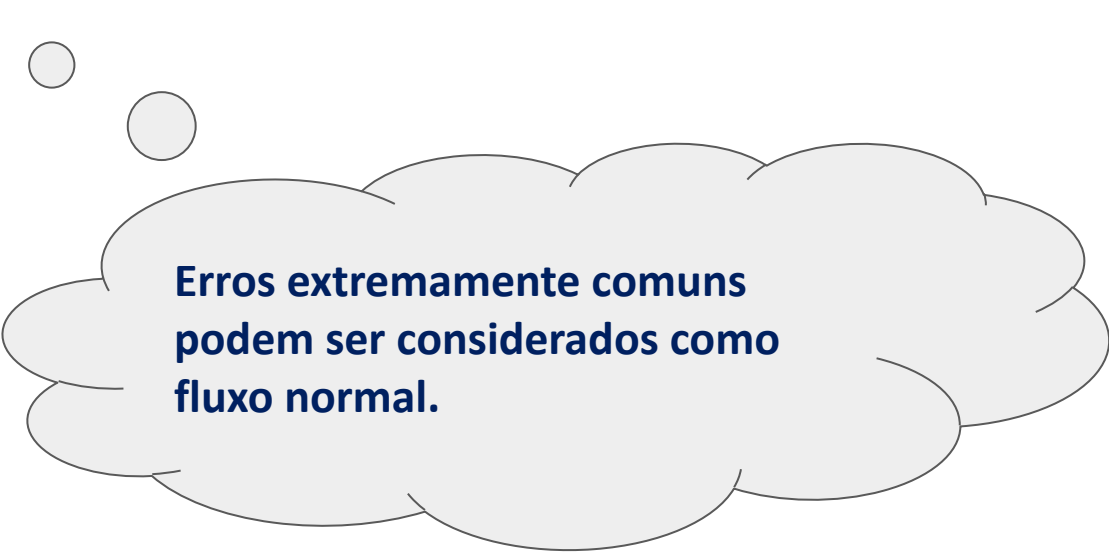
```
try
{
    conn.Close();
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.GetType().FullName);
    Console.WriteLine(ex.Message);
}
```

```
if (conn.State != ConnectionState.Closed)
{
    conn.Close();
}
```



Melhores práticas para exceções

5. Projete classes de forma que as exceções sejam evitadas ou minimizadas.



**Erros extremamente comuns
podem ser considerados como
fluxo normal.**

Melhores práticas para exceções

6. Lance exceções ao invés de somente retornar um status code.
7. Somente crie novas classes de exceções, caso as pré-definidas não satisfaçam as necessidades do código.
8. Crie classes com a terminação *Exception* e derivadas diretamente da classe base **Exception**.
9. Utilize no mínimo os construtores já definidos na classe base.

Melhores práticas para exceções

10. Escreva mensagens de erros claras e sucintas.
11. É uma boa prática incluir strings traduzidas de acordo com a linguagem do usuário da aplicação através de ***sattelites assemblies***.
12. Em exceções customizadas forneça propriedades adicionais conforme necessidade.



Melhores práticas para exceções

13. Utilize o comando ***throw*** para que o stack trace seja mais útil, pois o rastreamento começa a partir do lançamento até a captura da exceção.
14. Utilize métodos construtores de exceções.

```
FileNotFoundException NewFileIOException()  
{  
    string description = "My NewFileIOException Description";  
  
    return new FileNotFoundException(description);  
}
```



Melhores práticas para exceções

15. Restaure o estado da aplicação caso os métodos completem sua execução devido à exceções.

```
string withdrawalTrxID = from.Withdrawal(amount);  
try  
{  
    to.Deposit(amount);  
}  
catch  
{  
    from.RollbackTransaction(withdrawalTrxID);  
    throw;  
}
```



Melhores práticas para exceções

15. Restaure o estado da aplicação caso os métodos completem sua execução devido à exceções.

```
catch (Exception ex)
{
    from.RollbackTransaction(withdrawalTrxID);
    throw new TransferFundsException("Withdrawal failed.", innerException: ex)
    {
        From = from,
        To = to,
        Amount = amount
    };
}
```

Aula 5: Teste de unidade para exceções

Tratamento de Erro em C# com
Exceptions

Objetivos

1. Revisar o conceito e implementação de testes de unidade
2. Aprender a implementar testes de unidade para lançamento de exceções com MSTest V2, Xunit e NUnit.

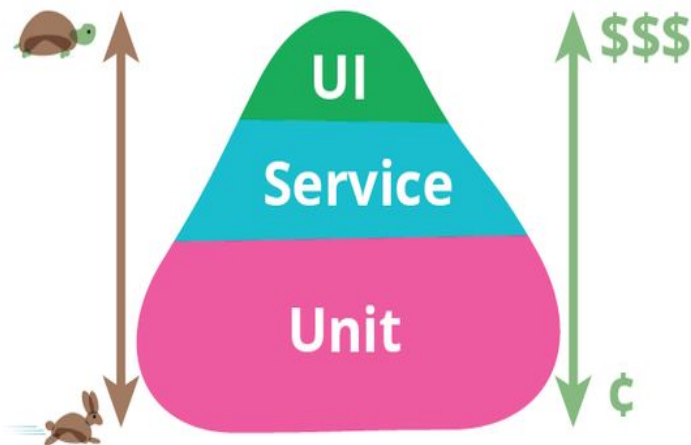
Etapa 1

Revisando testes de unidade

Tratamento de Erro em C# com
Exception



O que são testes unitários?



São testes que fazem verificação de unidades/componentes da aplicação, comparando a um retorno esperado com o retorno atual do código testado.

Estruturando um teste unitário

★ Padrão AAA

- Arrange
- Act
- Assert

```
public void Somar_ValoresPositivos_RetornarPositivo(){  
  
    //arrange  
    var x = 1;  
    var y = 2;  
    var resultadoEsperado = 3;  
    var calculadora = new Calculadora();  
  
    //act  
    var resultadoAtual = calculadora.soma(x,y);  
  
    //assert  
    Assert.Equal(resultadoEsperado, resultadoAtual);  
}
```

Etapa 2

Testando exceções

Tratamento de Erro em C# com
Exception



Como testar exceções?

- ★ O padrão e estruturação de testes não se modifica, contudo, o mecanismo de chamada da exceção pode variar de acordo com o framework utilizado.
 - MSTest V2
 - Xunit
 - NUnit



Usando MSTest V2

```
[TestMethod]
✓ public UnitMethod_Context_Return(){

    // Arrange : Configuração inicial e preparação de contexto de exceção
    var sut = new TestedClass();

    //Act: Chamar unidade da aplicação a ser testada a exceção
    var ex = Assert.ThrowsException<ExampleException>(() => sut.method());

    //Assert: Usar método de asserção contra o resultado da chamada do método
    Assert.AreEqual("Mensagem de erro esperada", ex.Message);

}
```



Usando Xunit

```
[Fact]
public UnitMethod_Context_Return(){

    // Arrange : Configuração inicial e preparação de contexto de exceção
    var sut = new TestedClass();

    //Act: Chamar unidade da aplicação a ser testada a exceção
    var ex = Assert.Throws<ExampleException>(() => sut.method());

    //Assert: Usar método de asserção contra o resultado da chamada do método
    Assert.Equal("Mensagem de erro esperada", ex.Message);
}
```



Usando Nunit

```
[Test]
public UnitMethod_Context_Return(){

    // Arrange : Configuração inicial e preparação de contexto de exceção
    var sut = new TestedClass();

    //Act: Chamar unidade da aplicação a ser testada a exceção
    var ex = Assert.Throws<ExampleException>(() => sut.method());

    //Assert: Usar método de asserção contra o resultado da chamada do método
    Assert.AreEqual("Mensagem de erro esperada", ex.Message);

    Assert.That(ex.Message, Is.EqualTo("Mensagem de erro esperada"));

    Assert.Throws(Is.TypeOf<ExampleException>()
        .And.Message.EqualTo("Mensagem de erro esperada"),
        () => sut.method());
}
```



Sugestão de projeto prático para uso de testes com exceções

1. Projeto Calculadora via console:

- Mínimo 4 operações
- Utilizar boas práticas para tratamento de exceções
- Criar projeto de teste de unidade para testar fluxos de sucesso e exceções.