Curso Profissional
Técnico de Gestão e Programação de Sistemas Informáticos

- PSI -
Programação e Sistemas de Informação

Módulo 11
Programação Orientada a Objetos Avançada

1. Introdução

Um dos objetivos a nível da planificação de uma aplicação é o desenvolvimento do código sem a ocorrência de erros aquando da sua execução.

No entanto, haverá sempre a possibilidade de ocorrer determinados erros não previstos.

1. Introdução

Este modulo tem como principal objetivo o tratamento estruturado de erros.

Os erros que ocorrem em sistemas informáticos podem dever-se, entre outros, a:

- ✓ Comportamento imprevisível do utilizador
- ✓ Falhas de hardware
- ✓ Problemas de conexão ou comunicação
- ✓ Falta de direito de acesso a recursos
- ✓ Erros de memória

1. Introdução

As linguagens de programação antigas utilizavam variáveis globais para identificar o erro:

- A lógica do programa ficava “entrelaçada” com o código do erro
- Códigos de erros numéricos não eram significativos
- Condições de erro eram pouco documentadas

As linguagens de programação mais recentes, encaram estas situações de maneira diferente.

1. Introdução

Consideremos o seguinte programa, o qual permite apresentar a soma de 2 números introduzidos pelo utilizador:

```
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void button1_Click(object sender, EventArgs e)
        {
                int a, b;
                a = Convert.ToInt32(textBox1.Text);
                b = Convert.ToInt32(textBox2.Text);
                textBox3.Text = Convert.ToString(a + b);
         }
    }
}
```

Supondo que o utilizador introduz os seguintes valores nas respetivas caixas de texto:

13

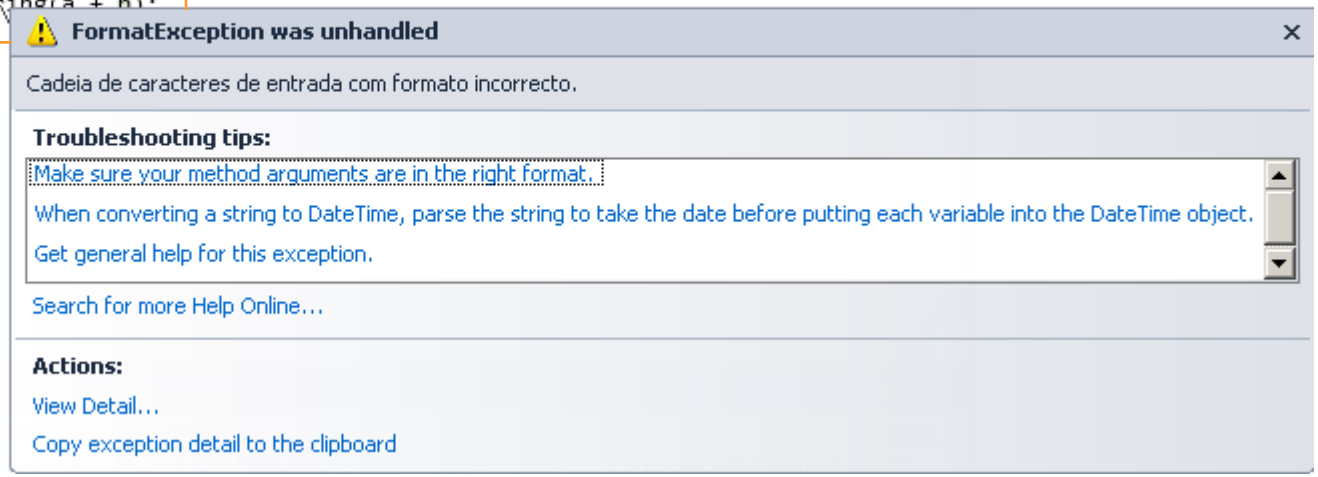
abc

1. Introdução

Uma vez que o segundo número introduzido não representa um valor numérico, ocorre um erro com o qual a aplicação não consegue lidar.

Desta foma, é apresentado uma informação, relatando ao utilizador que a aplicação “crashou”.

```
int a, b;
a = Convert.ToInt32(textBox1.Text);
b = Convert.ToInt32(textBox2.Text);
textBox3.Text = Convert.ToString(a + b);
```



1. Introdução

A mensagem de erro indica que foi lançada uma exceção quando o segundo parâmetro estava a ser convertido para um "*int*".

Tal como este erro, muitos outros podem ocorrer durante a execução das aplicações.

Em alguns casos é possível incluir verificações no código para prevenir a ocorrência de erros, mas muitas vezes isso iria complicar muito o código, e não seria possível contemplar todas as situações de erro.

O mecanismo de execução de exceções permite lidar com esse tipo de situações de forma mais simplificada.

2. Exceções

Linguagens de programação modernas sinalizam os erros utilizando exceções.

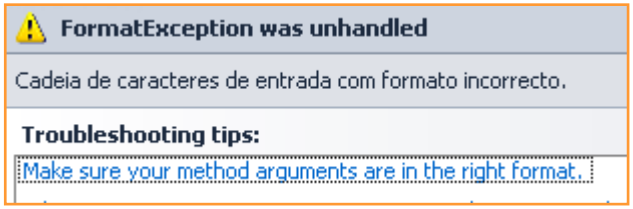
Exceções são objetos especiais que indicam um comportamento inesperado, interrompendo o fluxo normal de execução:

- ✓ Tratamento robusto dos erros
- ✓ Separação do código "principal" do programa e do código da manipulação de erros
- ✓ Diferentes tipos de erros produzem objetos de exceções de classes distintas
- ✓ As classes de exceções são organizadas numa hierarquia

2. Exceções

Tratamento de exceção

- ✓ Durante a execução de um programa, há sempre a possibilidade de ocorrer determinados eventos não previstos. Esses eventos são chamados de exceções.



- ✓ O tratamento de exceção é o mecanismo responsável pelo tratamento da ocorrência de condições que alteram o fluxo normal da execução de um programa.

- ✓ Uma falha num programa ocorre quando uma exceção não foi tratada ou não foram corrigidas as suas causas.

2. Exceções

O mecanismo de exceções permite lidar com erros que ocorrem durante a execução de programas, sem trazer grandes complicações para o código.

O objetivo principal dos mecanismos de tratamento de exceções é a construção de aplicações mais fiáveis, permitindo em caso de erro, terminar as aplicações de uma forma consistente, de modo a evitar a necessidade de recuperações e a corrupção de dados.

2. Exceções

Instruções utilizadas para o tratamento de exceções:

Instrução	Descrição
try	Bloco onde pode ocorrer o erro.
catch	Bloco onde o erro é tratado.
finally	Esta instrução é sempre executada. É normalmente utilizada quando se quer libertar um recurso, independentemente se ocorrer erro ou não.
throw	Esta instrução permite gerar as próprias exceções.

2. Exceções

Tendo em atenção o programa inicialmente apresentado, vamos analisá-lo com a inclusão de código para lidar com exceções:

```
private void button1_Click(object sender, EventArgs e)
{
        int a, b;
        a = Convert.ToInt32(textBox1.Text);
        b = Convert.ToInt32(textBox2.Text);
        textBox3.Text = Convert.ToString(a + b);
 }
```

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        int a, b;
        a = Convert.ToInt32(textBox1.Text);
        b = Convert.ToInt32(textBox2.Text);
        textBox3.Text = Convert.ToString(a + b);
    }
    catch
    {MessageBox.Show("Ocorreu um erro na … ");}
 }
```

O programa tem a mesma funcionalidade, mas agora, a não introdução de valores numéricos leva à mensagem de erro definida no programa, evitando que a aplicação “crash”.

2. Exceções

Exercício 01

a) Crie o programa anterior sem o mecanismo de exceções

b) Altere o programa aplicando o tratamento das exceções

2. Exceções

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        int a, b;
        a = Convert.ToInt32(textBox1.Text);
        b = Convert.ToInt32(textBox2.Text);
        textBox3.Text = Convert.ToString(a + b);
    }
    catch
    {MessageBox.Show("Ocorreu um erro na …");}
}
```

Mecanismo de exceções:

- ✓ O código em que podem ocorrer erros é contido num bloco *try*, seguido de um bloco *catch*. Isso faz com que, se for lançada uma exceção dentro do bloco *try*, o fluxo normal de execução é interrompido, passando a ser executado o código presente no bloco *catch*.

- ✓ O código dentro do bloco *catch* apresenta uma mensagem de erro, e a execução do programa continua no código a seguir aos blocos *try* e *catch*.

Neste caso, como não existe mais código, a aplicação termina (“sem crashar”).

2. Exceções

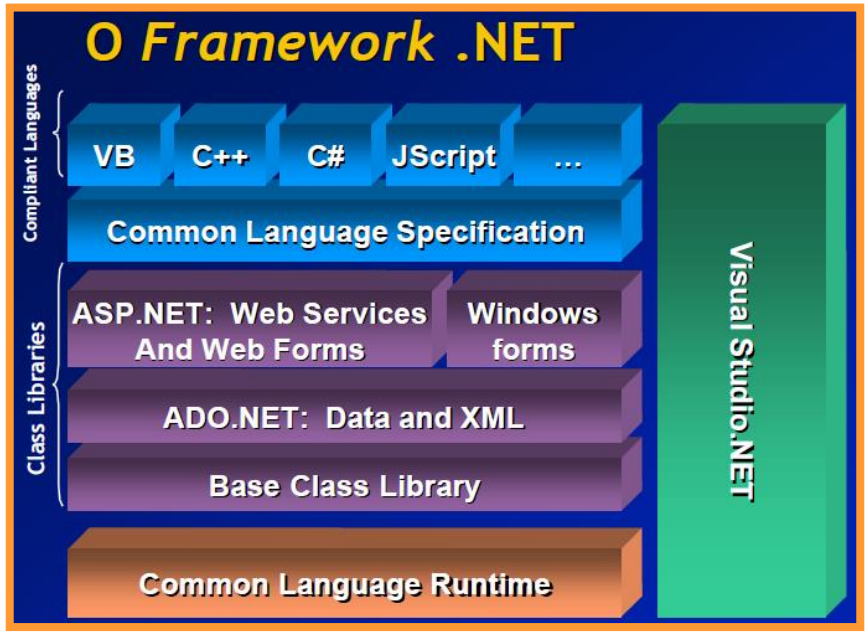
Exercício 02

Crie um programa simples que contemple a utilização de exceções.

3. Classes de Exceções

A *CLR (Common Language Runtime)* faz a gestão em tempo de execução do código (de acordo com o *Framework .Net*):

- ✓ Gestão automática de memória
- ✓ Gestão de *threads**
- ✓ Gestão de segurança
- ✓ Verificação de código
- ✓ Compilação de código



** Thread: divisão de um processo em duas ou mais tarefas que podem ser executadas em simultâneo.*

3. Classes de Exceções

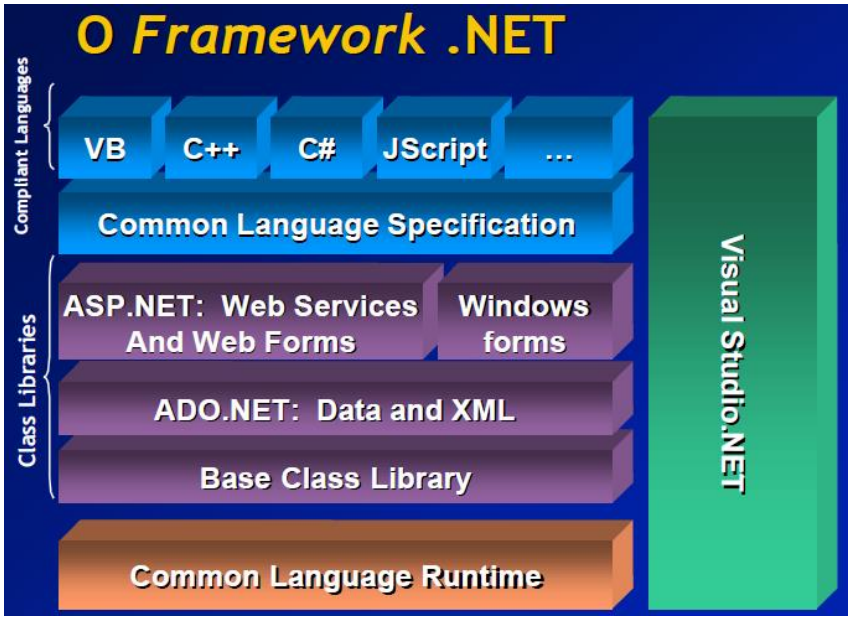
Assim sendo, em execução a *CLR (common language runtime)* é responsável pelo envio das *SystemException*.

Como exemplo de classes derivadas:

System.OutOfMemoryException

System.DivideByZeroException

System.OverFlowException



3. Classes de Exceções

Todas as exceções em *C#* são descendentes da classe `System.Exception`

As exceções do *CRL* derivam de `System.SystemException`

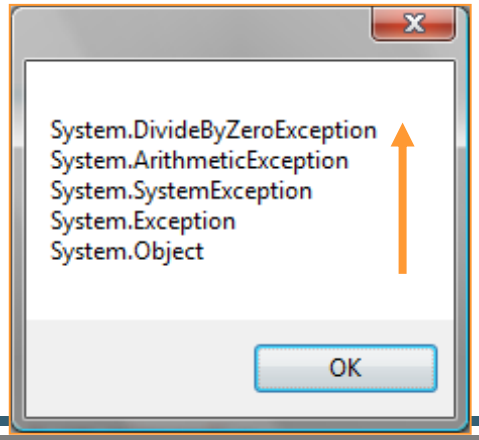
As exceções do utilizador são descendentes da classe `System.Exception`

```
catch (DivideByZeroException erro)
{
  MessageBox.Show(
    erro.GetType().ToString()+"\r\n"+
    erro.GetType().BaseType.ToString()+"\r\n"+
    erro.GetType().BaseType.BaseType.ToString()+"\r\n"+
    erro.GetType().BaseType.BaseType.BaseType.ToString()+"\r\n"+
    erro.GetType().BaseType.BaseType.BaseType.BaseType.ToString());
}
```

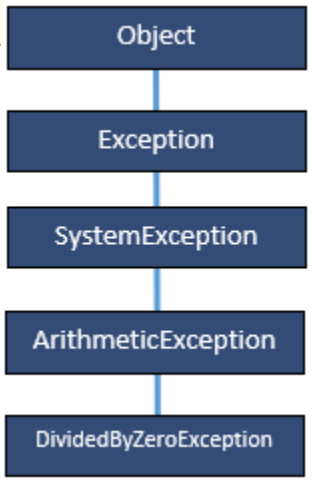
3. Classes de Exceções

```
catch (DivideByZeroException erro)
{
  MessageBox.Show(
    erro.GetType().ToString()+"\r\n"+
    erro.GetType().BaseType.ToString()+"\r\n"+
    erro.GetType().BaseType.BaseType.ToString()+"\r\n"+
    erro.GetType().BaseType.BaseType.BaseType.ToString()+"\r\n"+
    erro.GetType().BaseType.BaseType.BaseType.BaseType.ToString());
}
```

Output:



Esquema hierárquico:



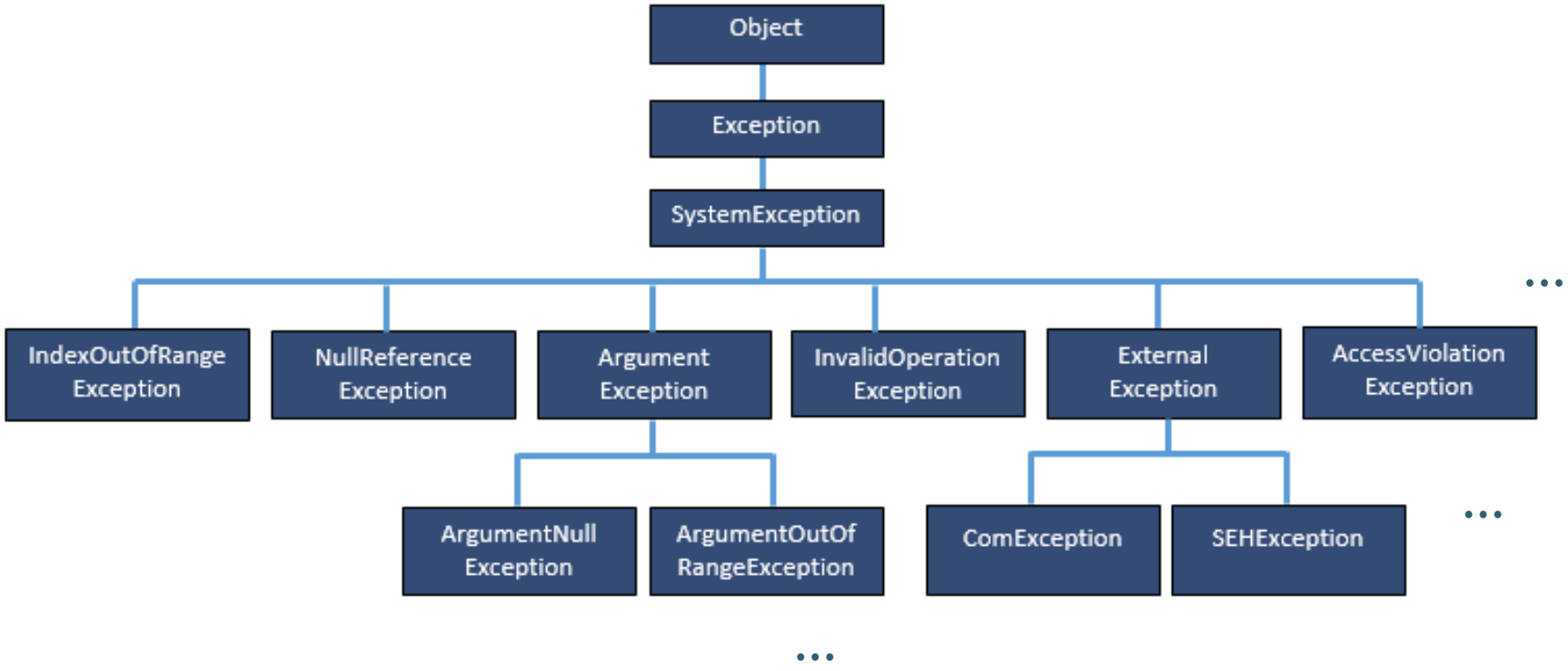
Hierarchy of Runtime Exceptions

The runtime has a base set of exceptions deriving from **SystemException** that it throws when executing individual instructions. The following table hierarchically lists the standard exceptions provided by the runtime and the conditions under which you should create a derived class.

Exception type	Base type	Description	Example
Exception	**Object**	Base class for all exceptions.	None (use a derived class of this exception).
SystemException	**Exception**	Base class for all runtime-generated errors.	None (use a derived class of this exception).
IndexOutOfRangeException	**SystemException**	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside its valid range: `arr[arr.Length+1]`
NullReferenceException	**SystemException**	Thrown by the runtime only when a null object is referenced.	`object o = null;` `o.ToString();`
AccessViolationException	SystemException	Thrown by the runtime only when invalid memory is accessed.	Occurs when interoperating with unmanaged code or unsafe managed code, and an invalid pointer is used.
InvalidOperationException	**SystemException**	Thrown by methods when in an invalid state.	Calling `Enumerator.GetNext()` after removing an `Item` from the underlying collection.
ArgumentException	**SystemException**	Base class for all argument exceptions.	None (use a derived class of this exception).
ArgumentNullException	**ArgumentException**	Thrown by methods that do not allow an argument to be null.	`String s = null;` `"Calculate".IndexOf (s);`
ArgumentOutOfRangeException	**ArgumentException**	Thrown by methods that verify that arguments are in a given range.	`String s = "string";` `s.Chars[9];`
ExternalException	**SystemException**	Base class for exceptions that occur or are targeted at environments outside the runtime.	None (use a derived class of this exception).
ComException	**ExternalException**	Exception encapsulating COM HRESULT information.	Used in COM interop.
SEHException	**ExternalException**	Exception encapsulating Win32 structured exception handling information.	Used in unmanaged code interop.

3. Classes de Exceções

Hierarquia de classes das exceções



Nota: este esquema apenas representa uma parte das exceções existentes.

4. Tratamento de Exceções

```
try {
// código que pode gerar excepções
}
catch                    {
// tratamento das excepções
}
finally {
// Este código é executado sempre
}
```

Bloco *Try*

O bloco `try` é um bloco de execução protegida onde deve ser implementado o fluxo principal do programa:

- ✓ O bloco `try` é obrigatório
- ✓ O *C#* tentará executar todas as instruções no bloco `try`
- ✓ Se nenhuma instrução gerar uma exceção, todas as instruções serão executadas

```
try
{
    int a, b;
    a = Convert.ToInt32(textBox1.Text);
    b = Convert.ToInt32(textBox2.Text);
    textBox3.Text = Convert.ToString(a + b);
}
```

4. Tratamento de Exceções

```
try {
// código que pode gerar excepções
}
catch                    {
// tratamento das excepções
}
finally {
// Este código é executado sempre
}
```

- ✓ Se ocorrer algum erro, a execução pula para os blocos `catch` ou `finally` (são opcionais); as restantes instruções no bloco `try`, são ignoradas

- ✓ Se nenhum `catch` captar a exceção (caso não esteja definida), a mesma é capturada pela *CLR* e a execução do aplicativo pode ser encerrada, dependendo do tipo de erro

4. Tratamento de Exceções

Bloco *Catch*

O bloco `try` é o bloco que manipula a exceção, sendo aqui onde devem ser implementados os tratamentos de erro:

- ✓ O bloco `catch` é opcional
- ✓ Um bloco `try` pode possuir vários blocos `catch`

```
try {
// código que pode gerar excepções
}
catch (Type1Exception e1) {
// tratamento das excepções tipo 1 e respectivas subclasses
}
catch (Type2Exception e2) {
// tratamento das excepções tipo 2 e respectivas subclasses
}
catch (Type3Exception e3) {
// tratamento das excepções tipo 3 e respectivas subclasses
}
finally {
// Este código é executado sempre
}
```

4. Tratamento de Exceções

- ✓ Os manipuladores devem ser escritos do mais específico para o mais genérico
- ✓ A classe mais genérica de erro é a Exception
- ✓ A exceção é consumida no bloco `catch`, ou seja, à priori apenas um catch é executado

```
try {
// código que pode gerar excepções
}
catch (Type1Exception e1) {
// tratamento das excepções tipo 1 e respectivas subclasses
}
catch (Type2Exception e2) {
// tratamento das excepções tipo 2 e respectivas subclasses
}
catch (Type3Exception e3) {
// tratamento das excepções tipo 3 e respectivas subclasses
}
finally {
// Este código é executado sempre
}
```

4. Tratamento de Exceções

Bloco *Finally*

No bloco `finally` são incluídas as intruções que devem necessariamente ser executadas, ocorram ou não erros:

- ✓ O bloco `finally` é opcional
- ✓ Um bloco `try` possui apenas um bloco finally

```
try {
// código que pode gerar excepções
}
catch (Type1Exception e1) {
// tratamento das excepções tipo 1 e respectivas subclasses
}
catch (Type2Exception e2) {
// tratamento das excepções tipo 2 e respectivas subclasses
}
catch (Type3Exception e3) {
// tratamento das excepções tipo 3 e respectivas subclasses
}
finally {
// Este código é executado sempre
}
```

4. Tratamento de Exceções

- ✓ Os blocos `finally` são utilizados para libertação de recursos como manipuladores de arquivos e conexões com bases de dados
- ✓ Numa exceção sem erros são executados os blocos `try` e `finally`
- ✓ Numa exceção com erros são executados os blocos `try` (parcialmente), *catch* (de acordo com o tipo do erro) e `finally` (sempre)

```
try {
// código que pode gerar excepções
}
catch (Type1Exception e1) {
// tratamento das excepções tipo 1 e respectivas subclasses
}
catch (Type2Exception e2) {
// tratamento das excepções tipo 2 e respectivas subclasses
}
catch (Type3Exception e3) {
// tratamento das excepções tipo 3 e respectivas subclasses
}
finally {
// Este código é executado sempre
}
```

4. Tratamento de Exceções

Nota:

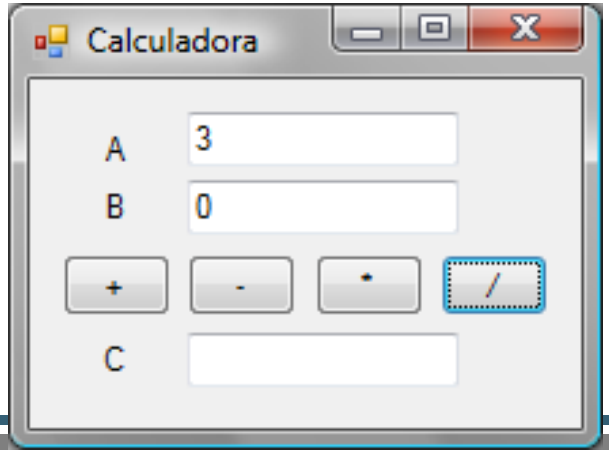
- ✓ É Importante que, havendo vários blocos `catch`, a ordem seja sempre da classe mais específica para a mais genérica
- ✓ Deverá ficar sempre por último "***Exception***", que é a classe genérica

4. Tratamento de Exceções

Exercício 03

Crie um programa que represente a imagem seguinte, utilizando 2 exceções:

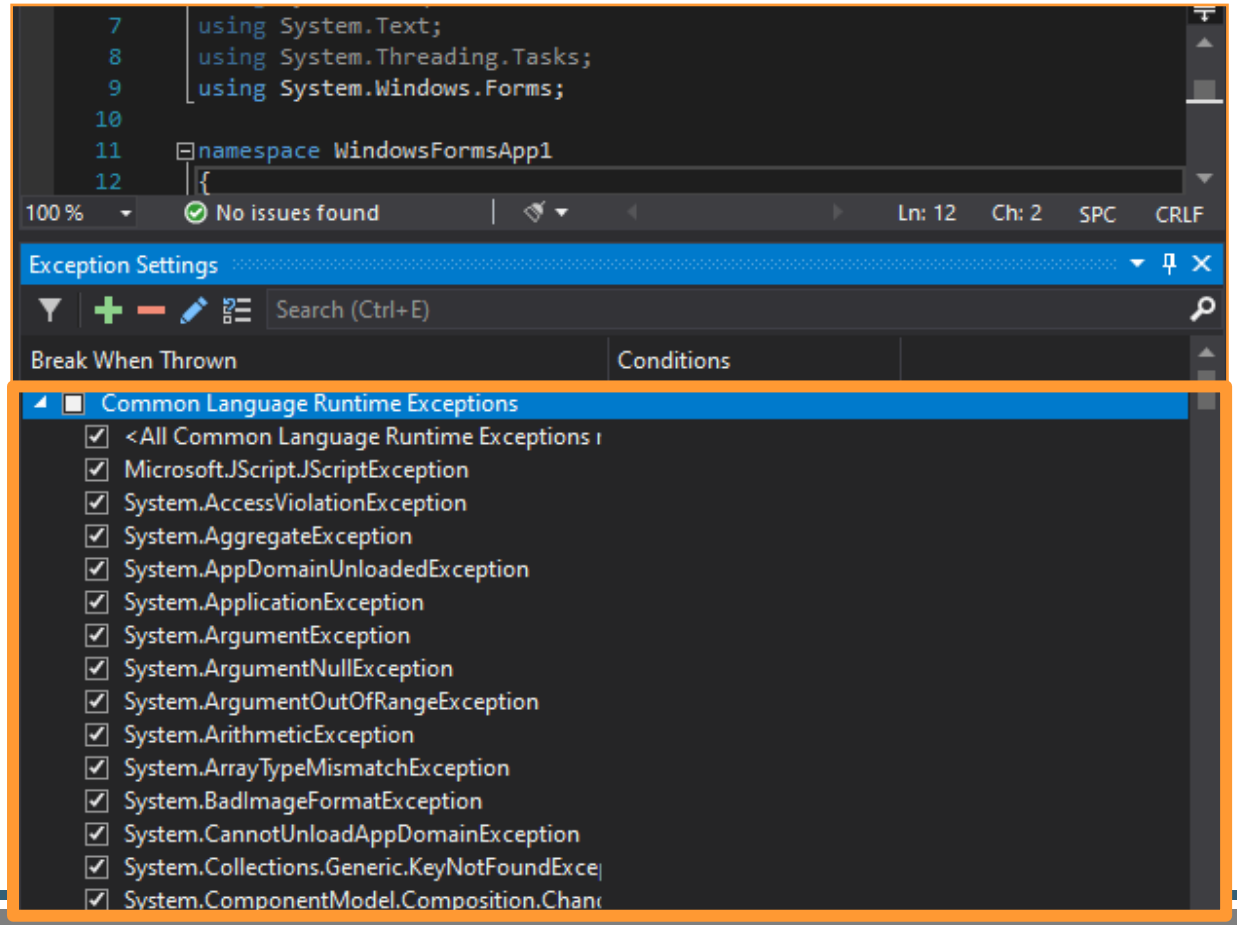
- ✓ responsável pelo tratamento da introdução de um valor não numérico
- ✓ responsável pelo tratamento da divisão de um número por zero



5. Principais Exceções

As exceções presentes na *CRL* (*Common Language Runtime*) podem ser consultadas, no *Visual C#*. Para isso, pode-se socorrer da combinação das teclas de atalho

CRT + ALT + E.



Trabalho de Projeto 01

6. Instrução *throw*

Esta instrução permite lançar uma exceção em situações que não são previstas, e que por si só, não constituem um erro.

Exemplo: pedir ao utilizador para inserir apenas valores pares.

```
int numero = Convert.ToInt32(textBox1.Text);
if ((numero % 2) != 0)
    throw new Exception("Digite apenas números pares");
else
    MessageBox.Show ("O número é válido");
```

6. Instrução *throw*

Exemplo: pedir ao utilizador para inserir apenas valores pares.

```
int numero = Convert.ToInt32(textBox1.Text);
if ((numero % 2) != 0)
    throw new Exception("Digite apenas números pares");
else
    MessageBox.Show ("O número é válido");
```

Esta instrução tem de ser colocada dentro do bloco *try*. E, para que funcione, é necessário definir o bloco *Catch* com a exceção pretendida e a definição de uma mensagem, para que seja apresentada a mensagem predefinida.

```
try {
    int numero = Convert.ToInt32(textBox1.Text);
    if ((numero % 2) != 0)
        throw new Exception("Digite apenas números pares");
    else
        MessageBox.Show ("O número é válido");
}

catch (Exception erro)
{
    MessageBox.Show(erro.Message);
}
```

A exceção definida na instrução *throw*, tem de ser a mesma que é chamada na instrução *catch*

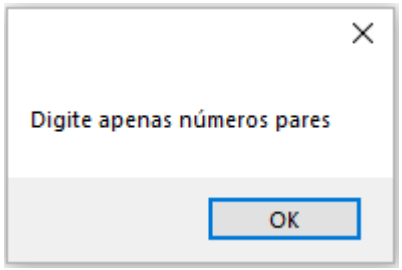
6. Instrução *throw*

```
try {
    int numero = Convert.ToInt32(textBox1.Text);
    if ((numero % 2) != 0)
        throw new Exception("Digite apenas números pares");
    else
        MessageBox.Show ("O número é válido");
}

catch (Exception erro)
{
    MessageBox.Show(erro.Message);
}
```

1) A exceção definida na instrução *throw*, tem de ser a mesma que é chamada na instrução *catch*

2) Definição da mensagem a aparecer, caso haja o tratamento da exceção

3) A caixa de mensagens vai apresentar a mensagem que foi pré-definida (na instrução *throw)*), caso seja desencadeada a exceção



4) Para que seja apresentada a mensagem pré-definida na instrução *throw (*em caso de tratamento da exceção), é necessário definir uma variável, no cabeçalho da instrução *catch*, e utilizá-la na instrução da *messagebox*

6. Instrução *throw*

Exercício 04

Tendo em conta as seguintes situações, lance as exceções necessárias de modo a que o utilizador cumpra o que lhe é solicitado.

Utilize ainda as exceções pré-definidas necessárias para previnir os possíveis erros.

a) O utilizador apenas pode inserir valores pares

b) O utilizador só pode inserir valores dentro de um determinado intervalo

c) O utilizador só pode digitar valores entre 1 e 12, por forma a escolher o mês correspondente (inclua a instrução *throw* dentro do bloco *try*)

d) Altere a aplicação da alínea anterior de modo a utilizar a instrução *throw* fora do bloco *try*

e) Ilustre uma outra situação por forma a provocar uma exceção

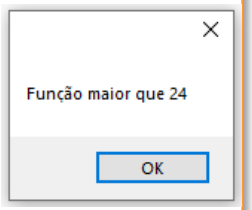
Nota: todas estas situações podem ser criadas dentro do mesmo programa

7. Criação de novas exceções

```
public partial class Form1 : Form
{
    private static string y;
    public Form1()
    {
        InitializeComponent();
    }
    class MaiorQue24Exception : Exception
    {
        public void mensagem_metodo()
        {
            MessageBox.Show("funcao maior que 24");
        }
    }

    class MenorQue12Exception : Exception
    {
        public string mensagem_propriedade
        {
           set {
                 y=value;
            }
            get {
             return y;
            }
        }
    }
```

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {

        int x = Int32.Parse(textBox1.Text);
        if (x > 24)
            throw new MaiorQue24Exception();
        else if (x < 12)
            throw new MenorQue12Exception();
    }
    catch (MaiorQue24Exception ver)
    {
        ver.mensagem_metodo();

        MessageBox.Show(ver.Message);
    }
    catch (MenorQue12Exception md)
    {
        md.mensagem_propriedade = "Atenção! você digitou um número menor que 12 ->";
        MessageBox.Show(md.mensagem_propriedade.ToString());
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

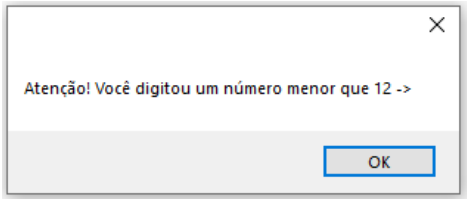
1 A instrução *throw* lança uma nova exceção (“*MaiorQue24Exception*”) criada pelo programador



2 Caso o utilizador digitar um valor superior a 24

3 A instrução *throw* lança outra exceção criada pelo programador (“*MenorQue12Exception*”)

4 Caso o utilizador digitar um valor inferior a 12



7. Criação de novas exceções

Exercício 05

Altere as alíneas a), b), c) e d) presentes no exercício 04 por forma a criar exceções.