**Lecture 6 - Exception Handling. Text formatting. Regular expressions**

The exception handling techniques. Strings processing.

The C# language's exception handling features provide a way to deal with any unexpected or exceptional situations that arise while a program is running. Exception handling uses the **try**, **catch**, and **finally** keywords to attempt actions that may not succeed, to handle failures, and to clean up resources afterwards. Exceptions can be generated by the common language runtime (CLR), by third-party libraries, or by the application code using the **throw** keyword.

In this example, a method tests for a division by zero, and catches the error. Without the exception handling, this program would terminate with a **DivideByZeroException was unhandled** error.

int SafeDivision(int x, int y)

{

try

{

return (x / y);

}

catch (System.DivideByZeroException dbz)

{

System.Console.WriteLine("Division by zero attempted!");

return 0;

}

}

## [Exceptions Overview](javascript:void(0)" \o "Click to collapse. Double-click to collapse all.)

Exceptions have the following properties:

* When your application encounters an exceptional circumstance, such as a division by zero or low memory warning, an exception is generated.
* Use a **try** block around the statements that might throw exceptions.
* Once an exception occurs within the **try** block, the flow of control immediately jumps to an associated exception handler, if one is present.
* If no exception handler for a given exception is present, the program stops executing with an error message.
* If a catch block defines an exception variable, you can use it to get more information on the type of exception that occurred.
* Actions that may result in an exception are executed with the **try** keyword.
* An exception handler is a block of code that is executed when an exception occurs. In C#, the **catch** keyword is used to define an exception handler.
* Exceptions can be explicitly generated by a program using the **throw** keyword.
* Exception objects contain detailed information about the error, including the state of the call stack and a text description of the error.
* Code in a **finally** block is executed even if an exception is thrown, thus allowing a program to release resources.

**Usage**

A [try](http://msdn.microsoft.com/en-us/library/0yd65esw.aspx) block is used by C# programmers to partition code that might be affected by an exception. Associated [catch](http://msdn.microsoft.com/en-us/library/0yd65esw.aspx) blocks are used to handle any resulting exceptions. A [finally](http://msdn.microsoft.com/en-us/library/zwc8s4fz.aspx) block contains code that is run regardless of whether or not an exception is thrown in the **try** block, such as releasing resources that are allocated in the **try** block. A **try** block requires one or more associated **catch** blocks, or a **finally** block, or both.

The following examples show a **try-catch** statement, a **try-finally** statement, and a **try-catch-finally** statement.

C#

try

{

// Code to try goes here.

}

catch (SomeSpecificException ex)

{

// Code to handle the exception goes here.

// Only catch exceptions that you know how to handle.

// Never catch base class System.Exception without

// rethrowing it at the end of the catch block.

}

C#

try

{

// Code to try goes here.

}

finally

{

// Code to execute after the try block goes here.

}

C#

try

{

// Code to try goes here.

}

catch (SomeSpecificException ex)

{

// Code to handle the exception goes here.

}

finally

{

// Code to execute after the try (and possibly catch) blocks

// goes here.

}

A **try** block without a **catch** or **finally** block causes a compiler error.

[Catch Blocks](javascript:void(0))

A **catch** block can specify the type of exception to catch. The type specification is called an *exception filter*. The exception type should be derived from [Exception](http://msdn.microsoft.com/en-us/library/system.exception.aspx). In general, do not specify [Exception](http://msdn.microsoft.com/en-us/library/system.exception.aspx) as the exception filter unless either you know how to handle all exceptions that might be thrown in the **try** block, or you have included a [throw](http://msdn.microsoft.com/en-us/library/1ah5wsex.aspx) statement at the end of your **catch** block.

Multiple **catch** blocks with different exception filters can be chained together. The **catch** blocks are evaluated from top to bottom in your code, but only one **catch** block is executed for each exception that is thrown. The first **catch** block that specifies the exact type or a base class of the thrown exception is executed. If no **catch** block specifies a matching exception filter, a **catch** block that does not have a filter is selected, if one is present in the statement. It is important to position **catch** blocks with the most specific (that is, the most derived) exception types first.

You should catch exceptions when the following conditions are true:

* You have a good understanding of why the exception might be thrown, and you can implement a specific recovery, such as prompting the user to enter a new file name when you catch a [FileNotFoundException](http://msdn.microsoft.com/en-us/library/system.io.filenotfoundexception.aspx)object.
* You can create and throw a new, more specific exception.

C#

int GetInt(int[] array, int index)

{

try

{

return array[index];

}

catch(System.IndexOutOfRangeException e)

{

throw new System.ArgumentOutOfRangeException(

"Parameter index is out of range.");

}

}

* You want to partially handle an exception before passing it on for additional handling. In the following example, a **catch** block is used to add an entry to an error log before re-throwing the exception.

C#

try

{

// Try to access a resource.

}

catch (System.UnauthorizedAccessException e)

{

// Call a custom error logging procedure.

LogError(e);

// Re-throw the error.

throw;

}

[Finally Blocks](javascript:void(0)" \o "Collapse)

A **finally** block enables you to clean up actions that are performed in a **try** block. If present, the **finally** block executes last, after the **try** block and any matched **catch** block. A **finally** block always runs, regardless of whether an exception is thrown or a **catch** block matching the exception type is found.

The **finally** block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects. See [using Statement (C# Reference)](http://msdn.microsoft.com/en-us/library/yh598w02.aspx) for more information.

In the following example, the **finally** block is used to close a file that is opened in the **try** block. Notice that the state of the file handle is checked before the file is closed. If the **try** block cannot open the file, the file handle still has the value **null** and the **finally** block does not try to close it. Alternatively, if the file is opened successfully in the **try** block, the **finally** block closes the open file.

C#

System.IO.FileStream file = null;

System.IO.FileInfo fileinfo = new System.IO.FileInfo("C:\\file.txt");

try

{

file = fileinfo.OpenWrite();

file.WriteByte(0xF);

}

finally

{

// Check for null because OpenWrite might have failed.

if (file != null)

{

file.Close();

}

}

**Regular expressions**

The Microsoft .NET Framework, which you can use with any .NET programming language such as C# (C sharp) or Visual Basic.NET, has solid support for regular expressions. The documentation of the regular expression classes is very poor, however. Read on to learn how to use regular expressions in your .NET applications. In the text below, I will use VB.NET syntax to explain the various classes. After the text, you will find a complete application written in C# to illustrate how to use regular expressions in great detail. I recommend that you download the source code, read the source code and play with the application. That will give you a clear idea how to use regexes in your own applications.

**single characters**

|  |  |
| --- | --- |
| Use | To match any character |
| [*set*] | In that set |
| [^*set*] | Not in that set |
| [*a–z*] | In the *a-z* range |
| [^*a–z*] | Not in the a-z range |
| . | Any except \n (new line) |
| \*char* | Escaped special character |

**control characters**

|  |  |  |
| --- | --- | --- |
| Use | To match | Unicode |
| \t | Horizontal tab | \u0009 |
| \v | Vertical tab | \u000B |
| \b | Backspace | \u0008 |
| \e | Escape | \u001B |
| \r | Carriage return | \u000D |
| \f | Form feed | \u000C |
| \n | New line | \u000A |
| \a | Bell (alarm) | \u0007 |
| \c *char* | ASCII control character | − |

**Non-ascii codes**

|  |  |
| --- | --- |
| Use | To match character with |
| \*octal* | 2-3 digit octal character code |
| \x *hex* | 2-digit hex character code |
| \u *hex* | 4-digit hex character code |

**character classes**

|  |  |
| --- | --- |
| Use | To match character |
| \p{*ctgry*} | In that Unicode category or block |
| \P{*ctgry*} | Not in that Unicode category or block |
| \w | Word character |
| \W | Non-word character |
| \d | Decimal digit |
| \D | Not a decimal digit |
| \s | White-space character |
| \S | Non-white-space char |

**quantifiers**

|  |  |  |
| --- | --- | --- |
| Greedy | Lazy | Matches |
| \* | **\*?** | 0 or more times |
| + | **+?** | 1 or more times |
| ? | **??** | 0 or 1 time |
| {*n*} | **{*n*}?** | Exactly *n* times |
| {*n,*} | **{*n,*}?** | At least *n* times |
| {*n,m*} | **{*n,m*}?** | From *n* to *m* times |

**anchors**

|  |  |
| --- | --- |
| Use | To specify position |
| ^ | At start of string or line |
| \A | At start of string |
| \z | At end of string |
| \Z | At end (or before \n at end) of string |
| $ | At end (or before \n at end) of string or line |
| \G | Where previous match ended |
| \b | On word boundary |
| \B | Not on word boundary |

**groups**

|  |  |
| --- | --- |
| Use | To define |
| (*exp*) | Indexed group |
| (?<*name*>*exp*) | Named group |
| (?<*name1-name2*>*exp*) | Balancing group |
| (?:*exp*) | Noncapturing group |
| (?=*exp*) | Zero-width positive lookahead |
| (?!*exp*) | Zero-width negative lookahead |
| (?<=*exp*) | Zero-width positive lookbehind |
| (?<!*exp*) | Zero-width negative lookbehind |
| (?>*exp*) | Non-backtracking (greedy) |

**inline options**

|  |  |
| --- | --- |
| Option | Effect on match |
| i | Case-insensitive |
| m | Multiline mode |
| n | Explicit (named) |
| s | Single-line mode |
| x | Ignore white space |

|  |  |
| --- | --- |
| Use | To |
| (?imnsx-imnsx) | Set or disable the specified options |
| (?imnsx-imnsx:*exp*) | Set or disable the specified options within the expression |

**backreferences**

|  |  |
| --- | --- |
| Use | To match |
| \*n* | Indexed group |
| \k<*name*> | Named group |

**alternation**

|  |  |
| --- | --- |
| Use | To match |
| *a* |*b* | Either *a* or *b* |
| (?(*exp*)  *yes* | *no*) | *yes* if *exp* is matched *no* if *exp* isn't matched |
| (?(*name*)  *yes* | *no*) | *yes* if *name* is matched *no* if *name* isn't matched |

**substitution**

|  |  |
| --- | --- |
| Use | To substitute |
| $*n* | Substring matched by group number *n* |
| ${*name*} | Substring matched by group *name* |
| $$ | Literal $ character |
| $& | Copy of whole match |
| $` | Text before the match |
| $' | Text after the match |
| $+ | Last captured group |
| $\_ | Entire input string |

**comments**

|  |  |
| --- | --- |
| Use | To |
| (?# *comment*) | Add inline comment |
| # | Add x-mode comment |

For detailed information and examples, see **http://aka.ms/regex**

To test your regular expressions, see **http://regexlib.com/RETester.aspx**

**supported unicode categories**

|  |  |
| --- | --- |
| Category | Description |
| Lu | Letter, uppercase |
| LI | Letter, lowercase |
| Lt | Letter, title case |
| Lm | Letter, modifier |
| Lo | Letter, other |
| L | Letter, all |
| Mn | Mark, nonspacing combining |
| Mc | Mark, spacing combining |
| Me | Mark, enclosing combining |
| M | Mark, all diacritic |
| Nd | Number, decimal digit |
| Nl | Number, letterlike |
| No | Number, other |
| N | Number, all |
| Pc | Punctuation, connector |
| Pd | Punctuation, dash |
| Ps | Punctuation, opening mark |
| Pe | Punctuation, closing mark |
| Pi | Punctuation, initial quote mark |
| Pf | Puntuation, final quote mark |
| Po | Punctuation, other |
| P | Punctuation, all |
| Sm | Symbol, math |
| Sc | Symbol, currency |
| Sk | Symbol, modifier |
| So | Symbol, other |
| S | Symbol, all |
| Zs | Separator, space |
| Zl | Separator, line |
| Zp | Separator, paragraph |
| Z | Separator, all |
| Cc | Control code |
| Cf | Format control character |
| Cs | Surrogate code point |
| Co | Private-use character |
| Cn | Unassigned |
| C | Control characters, all |

For named character set blocks (e.g., Cyrillic), search for "supported named blocks" in the MSDN Library.

**regular expression operations**

Class: System.Text.RegularExpressions.Regex

**Pattern matching with Regex objects**

|  |  |
| --- | --- |
| To initialize with | Use constructor |
| Regular exp | Regex(String) |
| + options | Regex(String, RegexOptions) |
| + time-out | Regex(String, RegexOptions,  TimeSpan) |

**Pattern matching with static methods**

Use an overload of a method below to supply the regular expression and the text you want to search.

**Finding and replacing matched patterns**

|  |  |
| --- | --- |
| To | Use method |
| Validate match | Regex.IsMatch |
| Retrieve single match | Regex.Match (first)  Match.NextMatch (next) |
| Retrieve all matches | Regex.Matches |
| Replace match | Regex.Replace |
| Divide text | Regex.Split |
| Handle char escapes | Regex.Escape  Regex.Unescape |

**Getting info about regular expression patterns**

|  |  |
| --- | --- |
| To get | Use Regex API |
| Group names | GetGroupNames GetGroupNameFromNumber |
| Group numbers | GetGroupNumbers  GetGroupNumberFromName |
| Expression | ToString |
| Options | Options |
| Time-out | MatchTimeOut |
| Cache size | CacheSize |
| Direction | RightToLeft |

[Miscellaneous Constructs](javascript:void(0))

Miscellaneous constructs either modify a regular expression pattern or provide information about it. The following table lists the miscellaneous constructs supported by the .NET Framework.

|  |  |  |
| --- | --- | --- |
| **Construct** | **Definition** | **Example** |
| **(?imnsx-imnsx)** | Sets or disables options such as case insensitivity in the middle of a pattern. For more information, see [Regular Expression Options](http://msdn.microsoft.com/en-us/library/yd1hzczs(v=vs.110).aspx). | \bA(?i)b\w+\b matches "ABA", "Able" in "ABA Able Act" |
| **(?#** *comment***)** | Inline comment. The comment ends at the first closing parenthesis. | \bA(?#Matches words starting with A)\w+\b |
| **#** [to end of line] | X-mode comment. The comment starts at an unescaped **#** and continues to the end of the line. | (?x)\bA\w+\b#Matches words starting with A |

**Questions**

1. What datatype describe string?
2. What operators and methods for string manipulation do you know?
3. What are the regular expressions ?
4. What are the **Regex Templates?**
5. Please name Regex methods and their meanings.