SoftServe
Ability. Agility. Advantage.

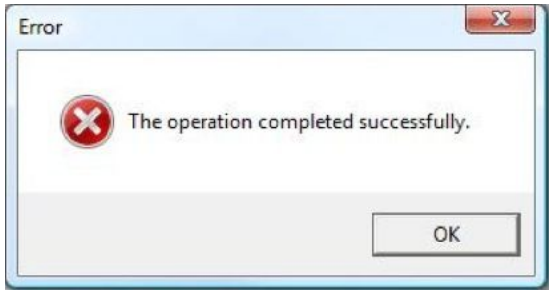
Structured Exceptions Handling in .NET

V'yacheslav Koldovskyy
SoftServe University
2014

Contents

1. Introduction to structured exception handling

Main task – correct operation of the application



- There are possible situations during the application execution when predetermined plan of actions may be changed
- Developer should provide ways to ensure correct execution despite possible errors

There are different kinds of errors reactions on which may be different and some may be corrected and some – don't:

- **Software errors** created by developer like reading of non-initialized variable;
- **System errors and failures** with resources, like memory exhaustion and file read errors;
- **User errors** like incorrect data input.

Obsolete check-based method

- Obsolete error handling method is based on multiple checks of input data and operation return codes.
- Drawbacks:
  - difficulties;
  - bloated code;
  - unreliable.

```
int IOResult = ReadFileWithIOResult("somefile.txt");
if (IOResult != 0)
{
    // Exception here, action required
}
else
{
    // File read successfully continuing normal execution
}
```

Structured exception handling

- Modern way to handle errors provides using of special mechanism – structured exception handling which is the part of programming language
- Exception is an event which happens during software execution and changes normal way of code execution
- Exceptions in .NET Framework are instances of classes inherited from base class Exception. Only instances of this class and inherited classes may participated in structured exception handling.

2. Construct «try..catch»

Simplest "try..catch" constuct

```
try
{
    // Code which may result in exception

}
catch
{
    // Code executed only in case of exception
}
```

"try..catch" construct with specific exception

```
try
{
    // Code which may result in exception
}
catch (DivideByZeroException)
{
    // Code executed in case of exception
}
```

Cascade sections of catch

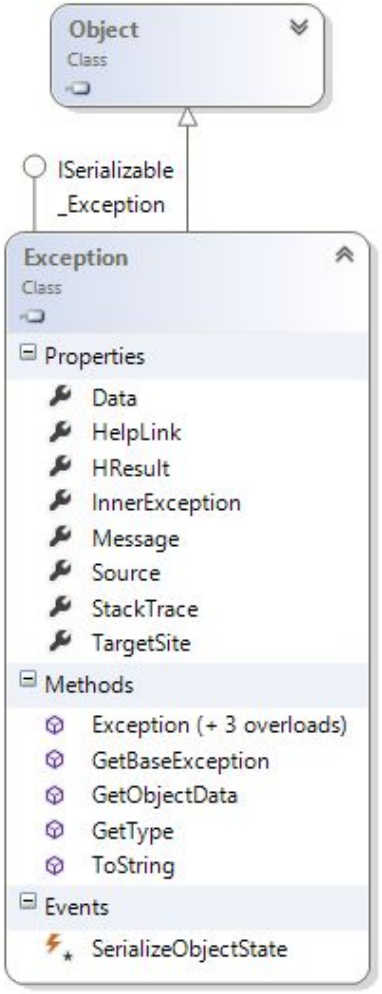
```
try
{
    // Code which may result in exception
catch (DivideByZeroException)
{
    // Code executed in case of exception type DivideByZeroException
}
catch (Exception)
{
    // Code executed in case of exception type Exception
    // Means "any exception"
}
```

"try..catch" construct with instance of exception

```
try
{
    // Code which may result in exception
}
catch (Exception e)
{
    // Code executed in case of exception
    // Using object e to get access to properties of exception
    Console.WriteLine(e.Message);
    // Re-rising same exception
    throw;
}
```

3. «Exception» class and exception hierarchy in.NET Framework

Exception class

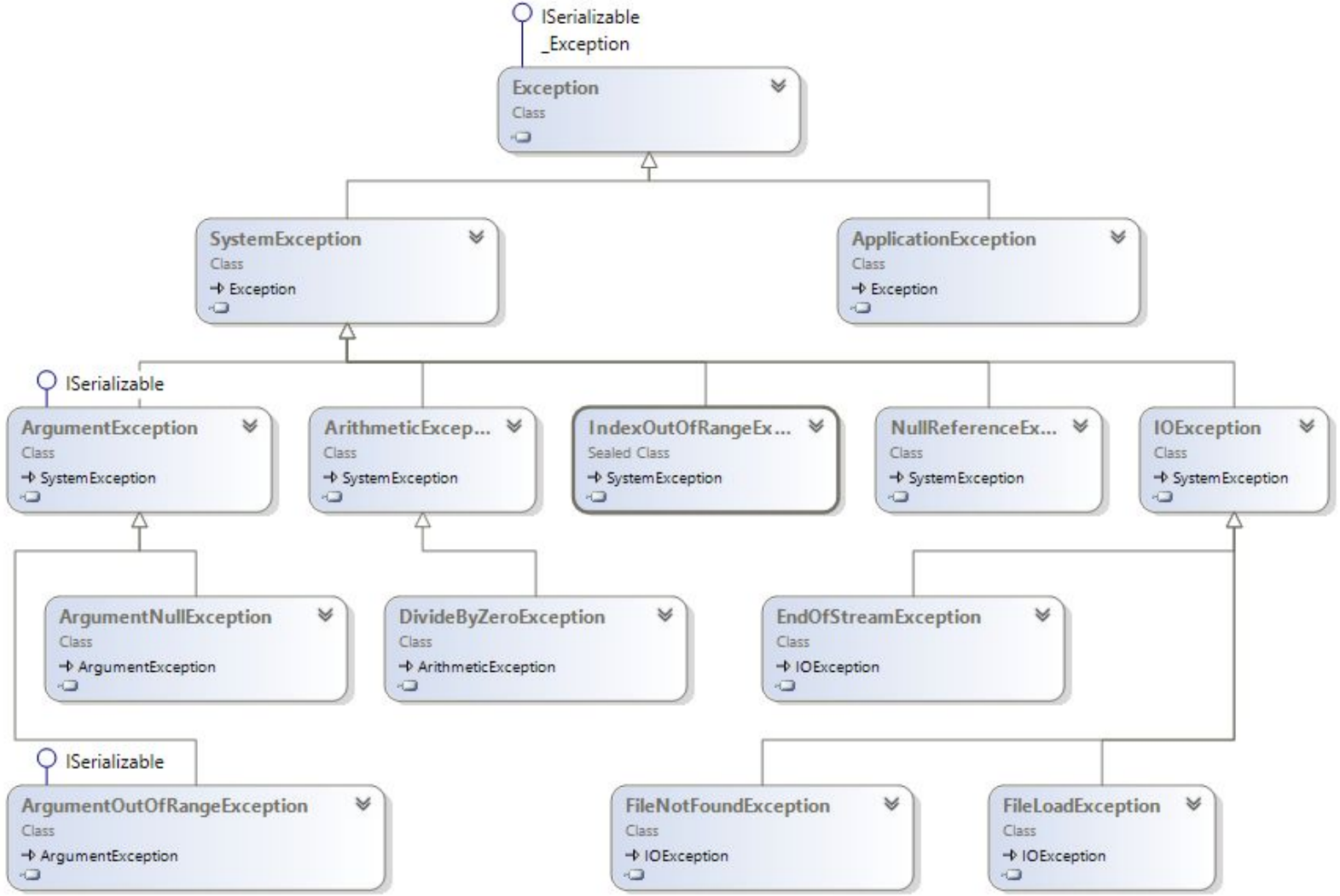


Exception is a base class for all exceptions исключений

Important properties:

- **Message** – user-oriented message about error
- **Source** – name of an error source (application or object)
- **InnerException** – inner exception (if called from other)
- **StackTrace** – call stack to the point of exception call
- **TargetSite** – method name which raised an exception
- **HelpLink** – URL-address to information about exception
- **Data** – dictionary with additional information with exception (IDictionary)

Exception hierarchy in .NET Framework



4. Exception throwing and re-rising

Exception throwing

```
public static void Demo(string SomeRequiredArg)
{
    // Check if some required argument is null
    if (SomeRequiredArg == null)
    {
        // Exception throwing
        throw new ArgumentNullException("Argument SomeRequiredArg is null");
    }
}
```

Exception re-rising

```
try
{
    // Code which may rise an exception
}
catch (Exception e)
{
    // Exception handling code
    // Using object e to get access to exception properties
    Console.WriteLine(e.Message);
    // Rising same exception again
    throw;
}
```

5. Creating own exceptions

Exception declaration

It is recommended to create own exceptions based on class ApplicationException.

Simplest declaration:

```
class SampleException: ApplicationException { };
```

To declare specific exceptions developers should create hierarchies of exceptions:

```
class SpecificSampleException: SampleException { };
```

MSDN recommendations for exception declarations

Minimal possible declaration for exception declaration described in MSDN requires use of Serializable attribute and definition of four constructors:

1) default constructor;
2) constructor which sets Message property;
3) constructor which sets Message and InnerException properties;
4) constructor for serialization.

```
[Serializable()]
public class InvalidDepartmentException : ApplicationException
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, System.Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) { }
}
```

6. Construct «try..finally»

Using finally

- «**try..finally**» used when it is required to guarantee execution of some code
- May be used together with catch

```
try
{
    // Code which may raise an exception
}
finally
{
    // Code which should be executed on any condition
}
```

7. Best practices for exception handling

Best practices for exception handling

- Do not catch general exceptions (do not use catch without parameters or **catch(Exception)**)
- Create own exceptions based on **ApplicationException** class but not on **SystemException**
- Do not use exceptions for application execution control flow as exception handling is heavy resource usage task. Exceptions should be used to manage errors only
- Do not mute exceptions which can't be handled in application context (system errors and failures).
- Do not raise general exceptions: **Exception**, **SystemException**, **ApplicationException**
- Do not generate reserved system exceptions: **ExecutionEngineException**, **IndexOutOfRangeException**, **NullReferenceException**, **OutOfMemoryException**
- Do not return an exception instance as a method return result instead of using **throw**.
- Do not create exceptions used only for debugging purposes. Do define debug-only exceptions use Assert.

8. References to additional sources

- MSDN recommendations for creating exceptions: http://msdn.microsoft.com/en-us/library/ms173163.aspx
- MSDN recommendation for exception generation: http://msdn.microsoft.com/en-us/library/ms182338.aspx
- Full hierarchy of Microsoft .NET Framework exceptions (code sample in comments): http://stackoverflow.com/questions/2085460/c-sharp-is-there-an-exception-overview

SoftServe
Ability. Agility. Advantage.

Contacts

Europe Headquarters

52 V. Velykoho Str.
Lviv 79053, Ukraine

Tel: +380-32-240-9090
Fax: +380-32-240-9080

E-mail: info@softservecom.com

US Headquarters

13350 Metro Parkway, Suite 302
Fort Myers, FL 33966, USA

Tel: 239-690-3111
Fax: 239-690-3116

E-mail: info@softservecom.com

www.softservecom.com

Thank you!