Exceptions Handling

Handling Errors during Program Execution

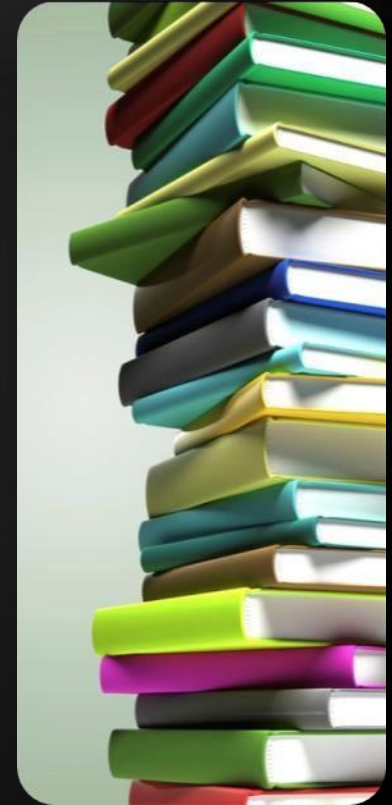**Avinash Tauro**

Arctech Info
www.arctechinfo.com

Table of Contents

What are Exceptions?

The Paradigm of Exceptions in OOP

What are Exceptions?

- **The exceptions in .NET Framework are classic implementation of the OOP exception model**

- **Deliver powerful mechanism for centralized handling of errors and unusual events**

- **Substitute procedure-oriented approach, in which each function returns error code**

- **Simplify code construction and maintenance**

- **Allow the problematic situations to be processed at multiple levels**

Handling Exceptions

Catching and Processing Errors

Handling Exceptions

- In C# the exceptions can be handled by the `try-catch-finally` construction

```
try
{
    // Do some work that can raise an exception
}
catch (SomeException)
{
    // Handle the caught exception
}
```

- `catch` blocks can be used multiple times to process different exception types

Handling Exceptions – Example

```
static void Main()
{
    string s = Console.ReadLine();
    try
    {
        Int32.Parse(s);
        Console.WriteLine(
            "You entered valid Int32 number {0}.", s);
    }
    catch (FormatException)
    {
        Console.WriteLine("Invalid integer number!");
    }
    catch (OverflowException)
    {
        Console.WriteLine(
            "The number is too big to fit in Int32!");
    }
}
```

Handling Exceptions

Live Demo

The System.Exception Class

- Exceptions in .NET are objects

- The System.Exception class is base for all exceptions in CLR

  - Contains information for the cause of the error or the unusual situation

    - Message – text description of the exception

    - StackTrace – the snapshot of the stack at the moment of exception throwing

    - InnerException – exception caused the current exception (if any)

Exception Properties – Example

```csharp
class ExceptionsTest
{

   public static void CauseFormatException()
   {

      string s = "an invalid number";
      Int32.Parse(s);
   }

   static void Main()
   {

      try
      {

         CauseFormatException();
      }
      catch (FormatException fe)
      {

         Console.Error.WriteLine("Exception caught: {0}\n{1}",
            fe.Message, fe.StackTrace);
      }
   }
}
```

Exception Properties

- **The Message property gives brief description of the problem**

- **The StackTrace property is extremely useful when identifying the reason caused the exception**

```
Exception caught: Input string was not in a correct
format.
    at System.Number.ParseInt32(String s, NumberStyles
style, NumberFormatInfo info)
    at System.Int32.Parse(String s)
    at ExceptionsTest.CauseFormatException() in
c:\console application1\exceptionstest.cs:line 8
    at ExceptionsTest.Main(String[] args) in
c:\console application1\exceptionstest.cs:line 15
```
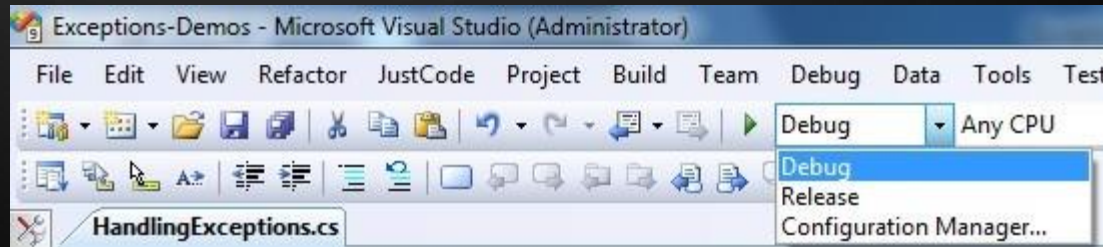
Exception Properties (2)

- File names and line numbers are accessible only if the compilation was in Debug mode

- When compiled in Release mode, the information in the property StackTrace is quite different:

```
Exception caught: Input string was not in a correct
format.
    at System.Number.ParseInt32(String s, NumberStyles
style, NumberFormatInfo info)
    at ExceptionsTest.Main(String[] args)
```
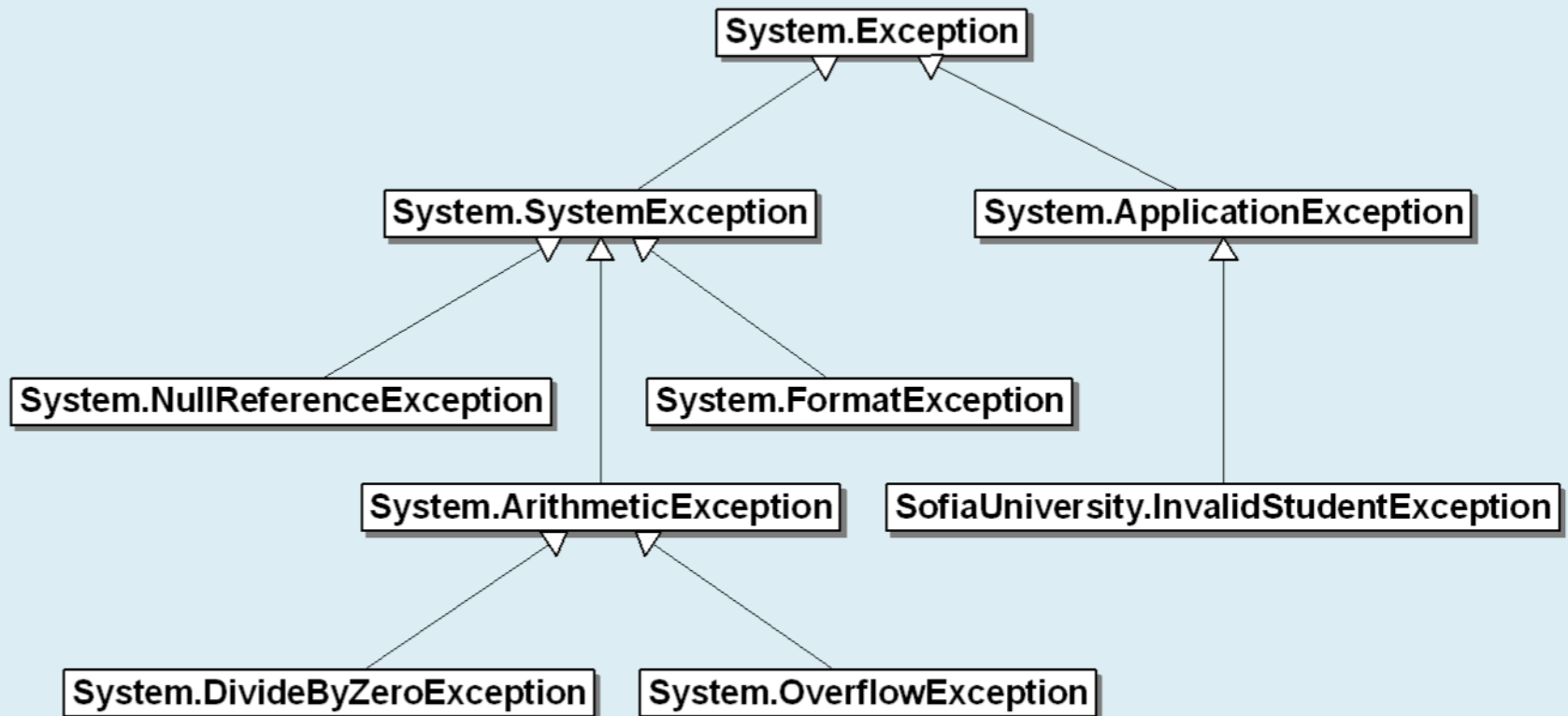
Exception Properties

Live Demo

The Hierarchy of Exceptions

Exception Hierarchy

Exceptions in .NET Framework are organized in a hierarchy

Types of Exceptions

- **All .NET exceptions inherit from System.Exception**

- **The system exceptions inherit from System.SystemException, e.g.**

  - **System.ArgumentException**

  - **System.NullReferenceException**

  - **System.OutOfMemoryException**

  - **System.StackOverflowException**

- **User-defined exceptions should inherit from System.ApplicationException**

Handling Exceptions

- When catching an exception of a particular class, all its inheritors (child exceptions) are caught too

- Example:

```
try
{
    // Do some works that can raise an exception
}
catch (System.ArithmeticException)
{
    // Handle the caught arithmetic exception
}
```
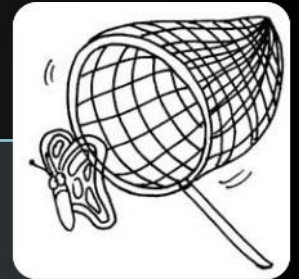
Handles ArithmeticException and its successors DivideByZeroException and OverflowException

```csharp
static void Main(string[] args)
{
    string s = Console.ReadLine();
    try
    {
        Int32.Parse(s);
    }
    catch (Exception)
    {
        Console.WriteLine("Can not parse the number!");
    }
    catch (FormatException)
    {
        Console.WriteLine("Invalid integer number!");
    }
    catch (OverflowException)
    {
        Console.WriteLine(
            "The number is too big to fit in Int32!");
    }
}
```

This should be last

Unreachable code

Unreachable code

Handling All Exceptions

- **All exceptions thrown by .NET managed code inherit the System.Exception exception**

- **Unmanaged code can throw other exceptions**

- **For handling all exceptions (even unmanaged) use the construction:**

```
try
{
    // Do some works that can raise any exception
}
catch
{
    // Handle the caught exception
}
```

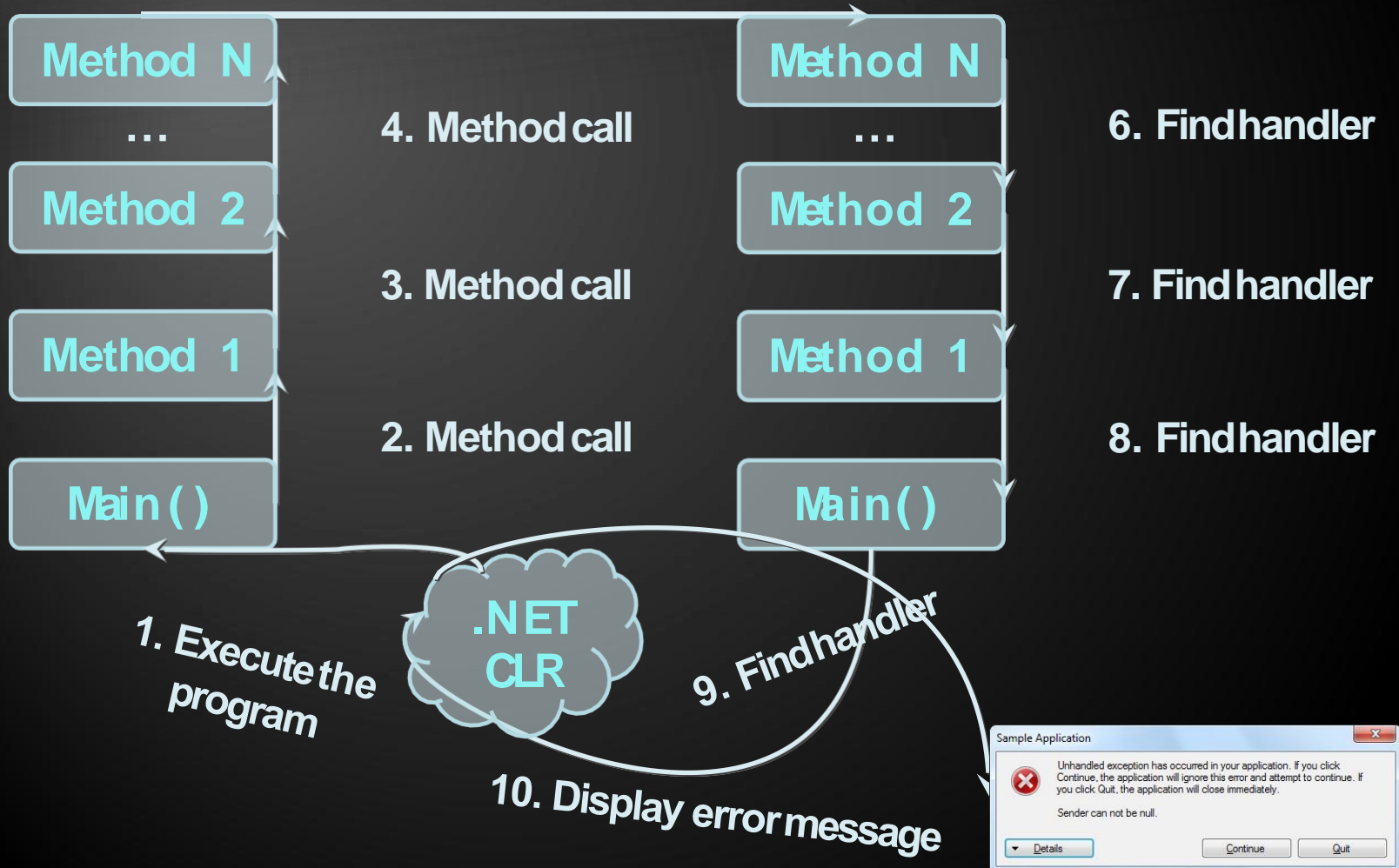# Throwing Exceptions

Throwing Exceptions

- **Exceptions are thrown (raised) by `throw` keyword in C#**
  - **Used to notify the calling code in case of error or unusual situation**
- **When an exception is thrown:**
  - **The program execution stops**
  - **The exception travels over the stack until a suitable `catch` block is reached to handle it**
- **Unhandled exceptions display error message**

How Exceptions Work?

5. Throw an exception

Method N	4. Method call	Method N	6. Find handler
...		...	
Method 2	3. Method call	Method 2	7. Find handler
Method 1	2. Method call	Method 1	8. Find handler
Main ( )		Main ( )	

.NET CLR

1. Execute the program

9. Find handler

10. Display error message

Sample Application

Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Sender can not be null.

Details   Continue   Quit

Using throw Keyword

⍰ **Throwing an exception with error message:**

```
throw new ArgumentException("Invalid amount!");
```

⍰ **Exceptions can take message and cause:**

```
try
{
    Int32.Parse(str);
}
catch (FormatException fe)
{
    throw new ArgumentException("Invalid number", fe);
}
```

⍰ **Note: if the original exception is not passed the initial cause of the exception is lost**

Re-Throwing Exceptions

Caught exceptions can be re-thrown again:

```
try
{
  Int32.Parse(str);
}
catch (FormatException fe)
{
  Console.WriteLine("Parse failed!");
  throw fe; // Re-throw the caught exception
}
```

```
catch (FormatException)
{
  throw; // Re-throws tha last caught exception
}
```

Throwing Exceptions – Example

```
public static double Sqrt(double value)
{
    if (value < 0)
        throw new System.ArgumentOutOfRangeException(
            "Sqrt for negative numbers is undefined!");
    return Math.Sqrt(value);
}
static void Main()
{
    try
    {
        Sqrt(-1);
    }
    catch (ArgumentOutOfRangeException ex)
    {
        Console.Error.WriteLine("Error: " + ex.Message);
        throw;
    }
}
```

Throwing Exceptions

Live Demo

Choosing Exception Type

- **When an invalid parameter is passed to a method:**
  - **ArgumentException, ArgumentNullException, ArgumentOutOfRangeException**
- **When requested operation is not supported**
  - **NotSupportedException**
- **When a method is still not implemented**
  - **NotImplementedException**
- **If no suitable standard exception class is available**
  - **Create own exception class (inherit Exception)**

Using Try-Finally Blocks

The try-finally Construction

- **The construction:**

```
try
{
    // Do some work that can cause an exception
}
finally
{
    // This block will always execute
}
```

- **Ensures execution of given block in all cases**
  - **When exception is raised or not in the try block**
- **Used for execution of cleaning-up code, e.g. releasing resources**

try-finally – Example

```
static void TestTryFinally()
{
    Console.WriteLine("Code executed before try-finally.");
    try
    {
        string str = Console.ReadLine();
        Int32.Parse(str);
        Console.WriteLine("Parsing was successful.");
        return; // Exit from the current method
    }
    catch (FormatException)
    {
        Console.WriteLine("Parsing failed!");
    }
    finally
    {
        Console.WriteLine("This cleanup code is always executed.");
    }
    Console.WriteLine("This code is after the try-finally block.");
}
```

Try-Finally

Live Demo

Exceptions: Best Practices

- `catch` blocks should begin with the exceptions lowest in the hierarchy and continue with the more general exceptions

  - Otherwise a compilation error will occur

- Each `catch` block should handle only these exceptions which it expects

  - Handling all exception disregarding their type is popular bad practice!

- When raising an exception always pass to the constructor good explanation message

Best Practices (2)

- **Exceptions can decrease the application performance**
  - **Throw exceptions only in situations which are really exceptional and should be handled**
  - **Do not throw exceptions in the normal program control flow (e.g.: on invalid user input)**
- **Some exceptions can be thrown at any time with no way to predict them, e.g.: System.OutOfMemoryException**

Summary

- **Exceptions provide flexible error handling mechanism in .NET Framework**

    - **Allow errors to be handled at multiple levels**

    - **Each exception handler processes only errors of particular type (and its childtypes)**

        - **Other types of errors are processed by other handlers**

    - **Unhandled exceptions cause error messages**

- **Try-finally ensures that given code block is always executed (even when an exception is thrown)**

Questions?

1. **Write a program that reads an integer number and calculates and prints its square root. If the number is invalid or negative, print "Invalid number". In all cases finally print "Good bye". Use try-catch-finally.**

2. **Write a method ReadNumber(int `start`, `int end`) that enters an integer number in given range [start..end]. If invalid number or non-number text is entered, the method should throw an exception. Based on this method write a program that enters 10 numbers:**

$$a_1, a_2, \dots a_{10}, \text{ such that } 1 < a_1 < \dots < a_{10} < 100$$

3. **Write a program that enters file name along with its full file path (e.g. C:\WINDOWS\win.ini), reads its contents and prints it on the console. Find in MSDN how to use** `System.IO.File.ReadAllText(…)`. **Be sure to catch all possible exceptions and print user-friendly error messages.**

4. **Write a program that downloads a file from Internet (e.g. http://www.devbg.org/img/Logo-BASD.jpg) and stores it the current directory. Find in Google how to download files in C#. Be sure to catch all exceptions and to free any used resources in the finally block.**