CHAPTER 5

C#

EXCEPTION HANDLING

Contents

1. Introduction
2. Exception Handling
3. Exception classes in C#
4. User defined Exception
5. Benefits of Exceptions

Introduction

- There are three categories of errors:
  - ***Syntax errors*** arise because the **rules of the language have not been followed**. They are detected by the compiler.
  - ***Runtime errors*** occur while the program is running if the environment detects an operation that is **impossible to carry out**.
  - ***Logical errors*** occur when a program **doesn't perform the way it was intended to**.

Exception

- An **Exception**
  - It is a **problem that arises during the execution of a program**.
  - A C# exception is **a response to an exceptional situation** that arises while a program is running, such as an attempt to **divide by zero**.
- **Exceptions** provide a way to **transfer control** from one part of a program to another

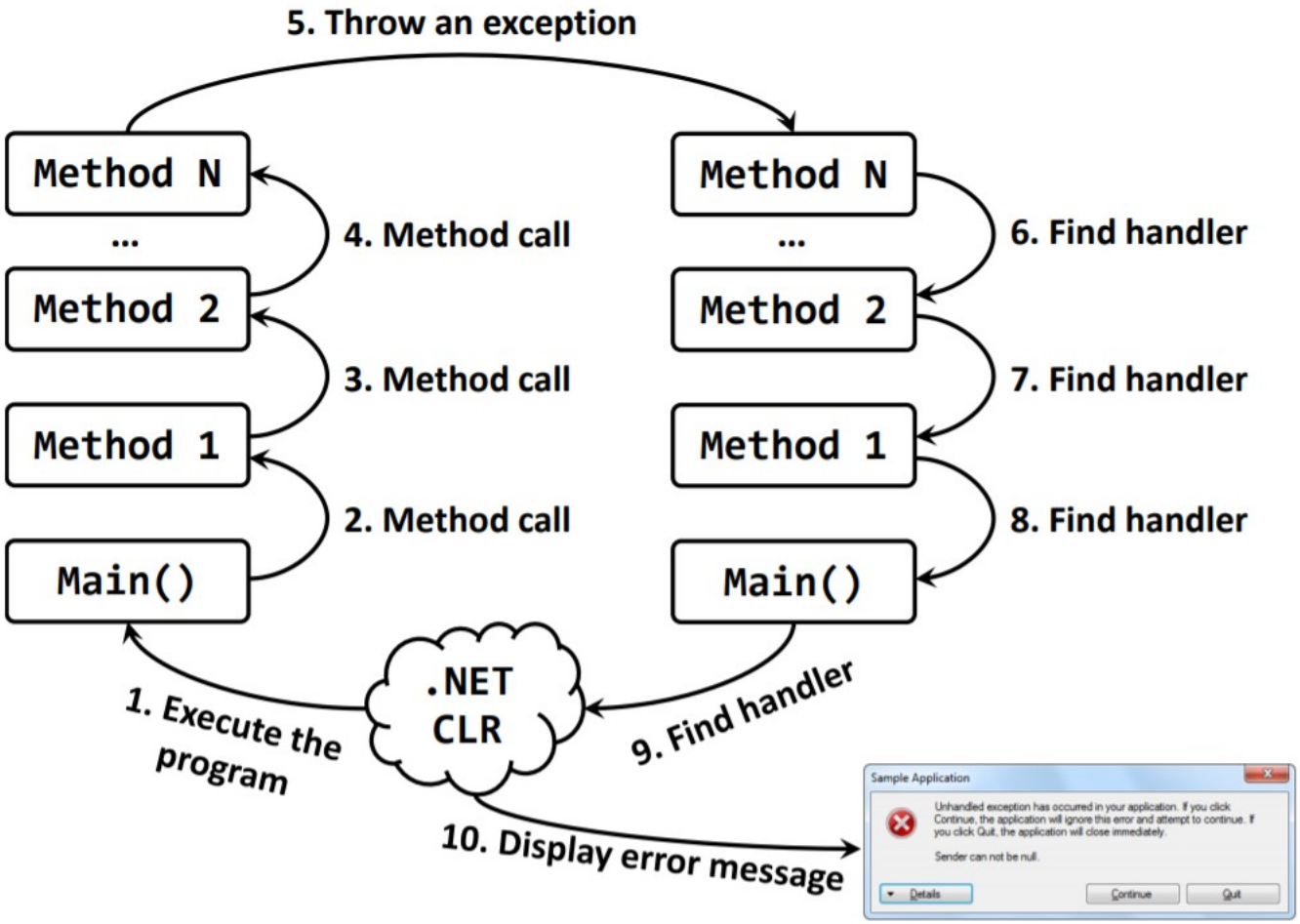
- **Exception in .NET** is an **object**, which signals an error or an event, which is not anticipated in the normal program flow.
  - When such unusual event takes place, the executing method **'throws**' a special object containing information about the **type of the error**, the place in **the program where the error occurred** as well as **the program state at the moment of the error**.
- Each exception in .NET contains the so-called **stack trace**, which gives information of where exactly the error occurred.

Exception Handling

- **Exception handling** is a mechanism, which allows exceptions to be **thrown** and **caught**.
  - This mechanism is **provided internally by the CLR** (Common Language Runtime).
  - Parts of the exception handling infrastructure are the language constructs in C# for throwing and catching exceptions.
  - CLR takes care to propagate each exception to the code that can handle it.

How Do Exceptions Handling Work?

- The exception handling mechanism follows a **reversed process**.
  1. When an exception is thrown, CLR begins searching an exception handler in the callstack starting from the method that has thrown the exception.
  2. This is repeated for each of the methods down the call-stack **until a handler is found** which catches the exception.
  3. If Main(…) is reached and **no handler is found**, **CLR catches the exception** and **usually displays an error message** (either in the console or in a special error dialog box).


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

- C# exception handling is built upon four keywords:
  1. **try**: A try block identifies a block of code for which particular exceptions will be activated.
     - **It's followed by one or more catch blocks**.
  2. **catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem.
     - The catch keyword indicates the **catching of an exception**.

3. **finally**: The finally block is used to execute a given set of statements, **whether an exception is thrown or not thrown**.
   - For **example**, if you open a file, it must be closed whether an exception is raised or not.
4. **throw**: A program throws an exception when a problem shows up. This is done using a throw keyword.

- **try** block
  - A **try** block contains code that requires common cleanup or exception-recovery operations.
  - The cleanup code should be put in a single **finally** block.
  - The exception recovery code should be put in **one or more catch** blocks.
    - Create one **catch** block for each kind of type you want to handle.
  - A **try** block must have at least one **catch** or **finally** block.

- **catch** block
  - A **catch** block contains code to execute in response to an exception.
  - If the code in a **try** block doesn't cause an exception to be thrown, the CLR will never execute the code in any of its catch blocks.
  - You may or may not specify a catch type in parenthesis after catch :
    - The catch type must be of type System.Exception or a type that derived from System.Exception
    - If there is no catch type specified, that catch block handles any exception. This is equivalent to having a catch block that specifies System.Exception as a catch type.

- Once the **catch** block that matches the exception is found, you have 3 choices:
  1. **Re-throw** the same exception, notifying the higher-up call stack of the exception
  2. **Throw** a different exception, giving richer exception information to code higher-up in the call stack
  3. Let the code continue from the bottom of the catch block
- In choices 1-2, an exception is thrown and code starts looking for a catch block whose type matches the exception thrown
- In choice 3, the **finally** block is executed
- You can also specify a variable name like **catch(Exception e)** to access information specific to the exception.

- **finally** block
  - C# provides the **finally** block, which is guaranteed to execute regardless of whether an exception occurs.
    - If the **try** block executes without throwing, the **finally** block executes.
    - If the **try** block throws an exception, the **finally** block still executes regardless of whether the exception is caught.
  - This makes the finally block ideal to **release resources from the corresponding try block**.

- Local variables in a **try** block cannot be accessed in the corresponding **finally** block, so variables that must be accessed in both should be declared before the try block.
- Placing the **finally** block before a **catch** block is a **syntax error**.
- A **try** block does not require a finally block, sometimes no clean-up is needed.
- A **try** block can have no more than **one finally** block.

Syntax

- Assuming a block will raise and exception, a method catches an exception using a combination of the **try and catch keywords**.
  - You can list down **multiple catch statements to catch different type of exceptions** in case your try block raises more than one exception in different situations.

```
try {
  // code that requires common cleanup or
  // exception-recovery operations
}
catch (InvalidOperationException) {
  //code that recovers from an InvalidOperationException
  // (or any exception type derived from it)
}
catch (SomeOtherException) {
  // code that recovers from an SomeOtherException
  // (or any exception type derived from it)
}
catch {
  // code that recovers from any kind of exception
  // when you catch any exception, you usually re-throw
  throw;
}
finally {
  // code that cleans up any operations started
  // within the try block. This code ALWAYS executes.
}
```

Exception Classes in C#

- C# exceptions are represented by classes.
  - The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class.
  - Some of the exception classes derived from the System.
- Exception class are the **System.ApplicationException** and **System.SystemException** classes.

- The **System.ApplicationException** class supports exceptions generated by application programs.
  - So the exceptions defined by the programmers should derive from this class.
- The **System.SystemException** class is the base class for all predefined system exception.

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

There are another Exception classes in C#, search about them !!

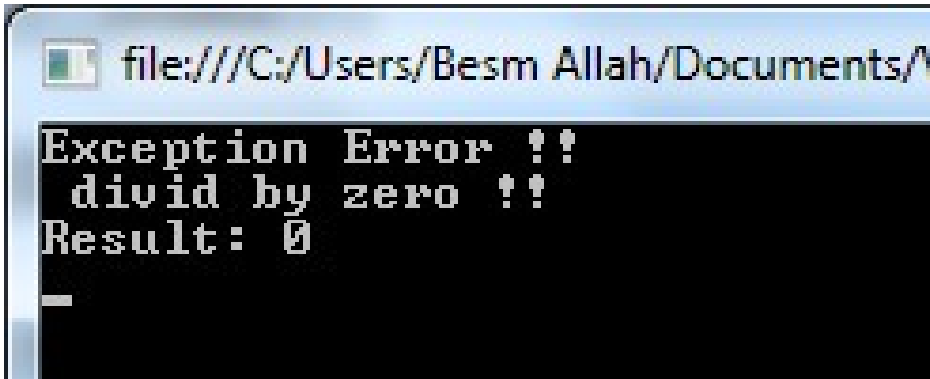
System.Exception properties

- Class Exception's properties are used to formulate error messages indicating a caught exception.
  - Property **Message** stores the error message associated with an Exception object.
  - Property **StackTrace** contains a string that represents the method-call stack.
- When an exception occurs, a programmer might use a **different error message** or indicate a new exception type.
  - The original exception object is stored in the InnerException property.

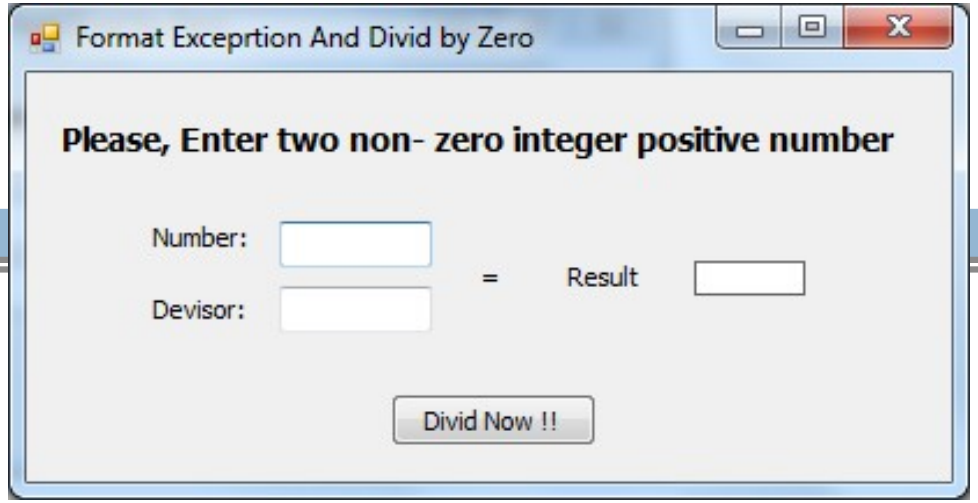
- Other properties:
  - **HelpLink** specifies the location of a help file that describes the problem.
  - **Source** specifies the name of the application or object that caused the exception.
  - **TargetSite** specifies the method where the exception originated.

Example 1:


```
1  class Program
2  {
3      public static void division(int num1, int num2)
4      {
5          float result=0.0f;
6          try
7          {
8           result = num1 / num2;
9          }
10         catch (DivideByZeroException e)
11         {
12             Console.WriteLine("Exception Error !! \n divid by zero !!");
13             // Console.WriteLine("Exception caught: {0}", e);
14         }
15         finally
16         {
17            Console.WriteLine("Result: {0} ", result);
18         }
19     }
20     static void Main(string[] args)
21     {
22         division(10,0);
23         Console.ReadLine();
24     }
25 }
```



Example 2:




```
public void divideNowBtn_Click(object sender, EventArgs e)
{
    float result=0.0f;
    try
    {
        result = int.Parse(txtNumber1.Text) / int.Parse(txtNumber2.Text);
    }
    catch (DivideByZeroException e)
    {
        Console.WriteLine("Exception Error !! \n divid by zero !!");
        // Console.WriteLine("Exception caught: {0}", e);
    }
    finally
    {
       Console.WriteLine("Result: {0} ", result);
    }
}
```

Choosing the Exception to throw

- When implementing your own methods, you should throw an exception when the method cannot complete its task.
- **Associating each type of malfunction** with an **appropriately named exception class** improves program clarity.
  1. What **Exception-derived type** are you going to throw?
  2. What **string message** are you going to pass to the exception type's constructor?

Do not catch everything?

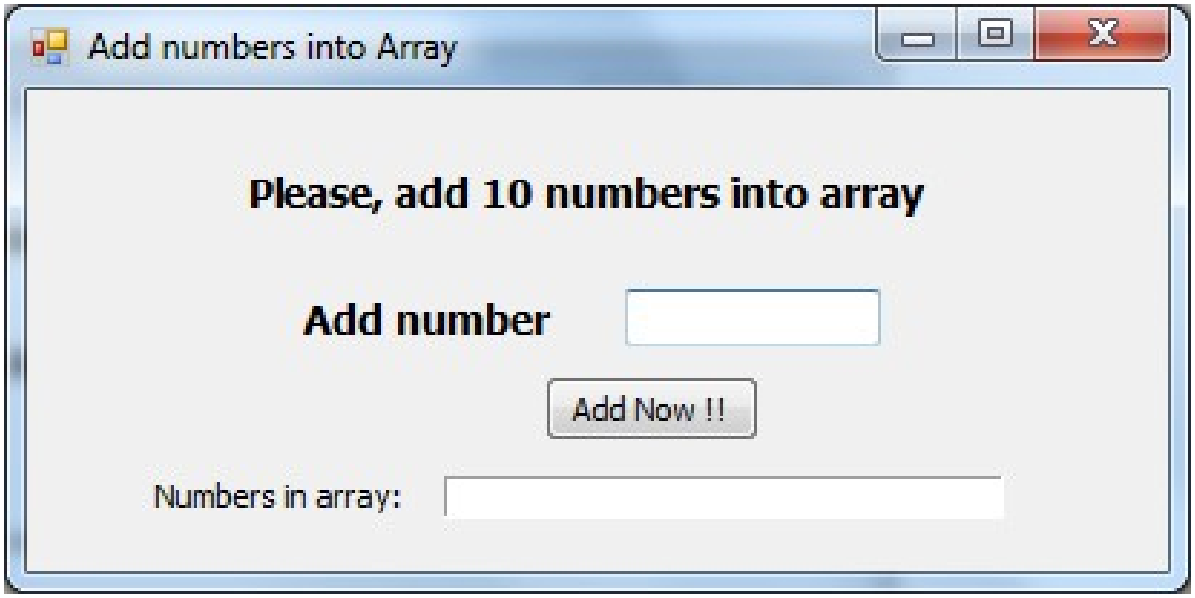

```
try {
  // code that might fail…
}
catch (Exception) {
  …
}
```

- How can you write code that can recover from all situations???
- A class library should never ever swallow all exceptions. The application should get a chance to handle the exception.
- You can catch all exceptions only if you are going to process it and re-throw it again.

Array in C#

- Declaring an Array
  - int [] my_array= new int[size];
- Array initialization
  - int [] my_array= {1,2,3,4};
- Access to the elements of an Array
  - We access the array elements directly using their **indices**. (0 to size-1)
  - My_array[index]=100;

Exercise 1:



Use
IndexOutOfRangeException
&
FormatException
&
Exception

User Defined Exceptions

- **User-defined exception** classes should derive directly or indirectly from class **Exception** of namespace System.
- Exceptions should be documented so that other developers will know how to handle them.
- User-defined exceptions should define three constructors:
  - A **parameterless** constructor
  - A constructor that receives a string argument (**the error message**)
  - A constructor that receives a string argument and an Exception argument (the error message and the inner exception object)

Example 3:


```
1  using System;
2  namespace ExceptionHandling
3  {
4      class NegativeNumberException:ApplicationException
5      {
6          public NegativeNumberException(string message)
7              // show message
8      }
9  }
```

```
if(value<0)
        throw new NegativeNumberException(" Use Only
Positive numbers");
```

Benefits of Exceptions

- The ability to keep **cleanup code** in a dedicated location and making sure this cleanup code will execute
- The ability to keep code that deals with **exceptional situations** in a central place
- The ability to **locate** and **fix bugs** in the code
- **Unified error handling**: all .NET Framework classes throw exceptions to handle error cases

- Exceptions also include a **stack trace** that tells you the path application took until the error occurred.
- You can also put any information you want in a **user-defined exception** of your own.
- The caller could **ignore the error** returned by a Win32 API, now the caller cannot continue with exceptions. If the application cannot handle the exception, CLR can terminate the application.

Exercise 2:

- Is the following code legal?

```
try {
    ...
} finally {
    ...
}
```

Exercise 3:

- What exception types can be caught by the following handler?

```
catch (Exception e) {
    ...
}
```

  - What is wrong with using this type of exception handler?

Exercise 4: Order of Exception

(A)

```
void function1()
{
    try
    {
        // code
    }
    catch(Exception ex)
    {
    }
    catch(Exception1 ex)
    {
    }
}
```

(B)

```
void function1()
{
    try
    {
        // code
    }
    catch(Exception1 ex)
    {
    }
    catch(Exception ex)
    {
    }

    // if no rethrow occurs
    // execution resumes her
}
```

Exercise 5: Adding Exceptions-1

- This program is throwing exception IndexOutOfRangeException.

```
static void Main(string[] args)
{
    string[] list = new string[5];
    list[0] = "Sunday";
    list[1] = "Monday";
    list[2] = "Tuesday";
    list[3] = "Wednesday";
    list[4] = "Thursday";

    for (int i = 0; i <= 5; i++)
    {
        Console.WriteLine(list[i].ToString());
    }
    Console.ReadLine();

}
```

Exercise 6: Adding Exceptions-2

- The given program is throwing **OverflowException**

```
static void Main(string[] args)
{
    int num1, num2;
    byte result;

    num1 = 30;
    num2 = 60;
    result = Convert.ToByte(num1 * num2);
    Console.WriteLine("{0} x {1} = {2}", num1, num2, result);
    Console.ReadLine();
}
```

Exercise 7: FormatExceptions

- Create a program that inputs miles driven and gallons used, and calculates miles per gallon.
  - The example should use exception handling to process the **FormatExceptions** that occur when converting the input strings to doubles. If invalid data is entered, display a message informing the user

Exercise 8: User defined Exception-1

- Write a program that takes a **positive integer** from the console and prints the **square root** of this integer. If the input is **negative** or **invalid** print **"Invalid Number"** in the console. In all cases print **"Good Bye".**

Exercise 9: User defined Exception-2

- Write a method ReadNumber(int start, int end) that reads an integer from the console in the range [start…end].
  - In case the input integer is not valid or it is not in the required range throw appropriate exception.
  - Using this method, write a program that takes 10 integers a1, a2, …, a10 such that 1 < a1 < … < a10 < 100.
  - (When invalid number is used we can throw Exception because there is no other exception that can better describe the problem. As an alternative we can define our own exception class called in a way that better describes the problem, e.g. **InvalidNumberException)**

Questions?

Thank You