



.NET Framework 4.6 and C # 6.0

Lesson 5:

Exception Handling in C#

Lesson Objectives



In this lesson we will cover the following :

- Understand the need for Exception Handling.
- Learn exception handling in C#. Use try, catch and finally blocks.
- Understand how to create user-defined exceptions.





What is an Exception?

Definition: An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a program.

Exceptions are notifications that some error has occurred in the program.

When an exception occurs you can ignore the exception or you can write code to deal with the exception, this is known as exception handling.

In this lesson you will learn the concepts of exceptions and exception handling.

C#'s approach to exception handling is a blend of and improvement on the exception handling mechanisms used by C++ and Java.

5.1: Exception and Exception Handling

Exception Handling in .NET



- While executing a program if a run-time error occurs, an exception is generated; this is usually referred to as an exception being thrown.
- An exception is an object that contains information about the runtime error which has occurred.
- We can use various techniques to act on an exception.
- In general, this is known as exception handling, and it involves writing code that will execute when an exception is thrown.

C#'s approach to exception handling is a blend of and improvement on the exception handling mechanisms used by C++ and Java.

Some exceptions are thrown by the .NET Framework to indicate a runtime error has occurred such as attempting to open a file that does not exist, or trying to convert a character string to a numeric datatype. You can also throw your own exceptions when business rules are broken, or to indicate that validation has failed, for example you can throw an exception when an invalid password is entered by a user, or when someone attempts to withdraw more than their daily limit from a bank account.

Programming with structured exception handling involves the use of four interrelated entities:

- Class type that represents details of the exception occurred.
- A member to throw an instance of the exception class to the caller.
- A block of code on the caller's side that invokes the exception-prone member.
- A block of code on the caller's side that processes (or catches) the exception.

Now that you have seen the basic concept of exceptions and exception handling, you are ready to implement exception handling in your code.

Exceptions are responses to abnormal or exceptional conditions that arise while a program executes. The common language runtime provides an exception handling model that is based on the representation of exceptions as objects, and the separation of program code and exception handling code into try block and catch block, respectively. There can be one or more catch blocks, each designed to handle a particular type of exception, or one block designed to catch a more specific exception than another block.

The exception handling is deeply ingrained in the .NET Runtime, and is therefore very common in C# code. It allows the code examples to be better.

Abnormalities Occurring

- What happens if the file cannot be opened?
- What happens if you cannot determine the length of the file?
- What happens if enough memory cannot be allocated?
- What happens if the read fails?
- What happens if the file cannot be closed?

```
readFile  
{  
    open the file;  
    find its size;  
    allocate memory;  
    read file into memory;  
    close the file;  
}
```

No matter how good your coding is, your programs always have to be able to handle possible errors. For example, in the middle of some complex processing your code discovers that it does not have permission to read a file, or while it is sending network requests the network goes down. In such exceptional situations, it is not enough for a method to simply return an appropriate error code – there might be 15 or 20 nested method calls, so what you really need the program to do is jump back up through all those 15 or 20 calls in order to exit the task completely and sort out the mess. C# has very good facilities to handle this kind of situation, through the mechanism known as *exception handling*.

5.3: System.Exception Class Overview



In C#, exceptions are represented by classes.

- All exception classes must be derived from the built-in exception class `Exception`.
 - `Exception` is part of the `System` namespace.
- From `Exception`, are derived classes that support two general exception categories defined in C#:
 - `SystemException` – Generated by C# runtime system.
 - `ApplicationException` – Generated by Application programs.

In C#, an exception is an object created (or *thrown*) when a particular exceptional error condition occurs. This object contains information that should help track down the problem. Although you can create our own exception classes (and you will be doing so later), .NET provides us with many predefined exception classes too.

The class hierarchy for exceptions is somewhat unusual in that most of these classes do not add any functionality to their respective base classes. However, in the case of exception handling, the usual reason for adding inherited classes is simply to indicate more specific error conditions, and there is often no need to override methods or add any new ones.

The generic exception class, `System.Exception` is derived from `System.Object`, as you would expect for a .NET class. In general, you should not throw generic `System.Exception` objects from your code, because they give no idea of the specifics of the error condition. There are two important classes in the hierarchy that are derived from `System.Exception`.

System.SystemException: This is for exceptions that are usually thrown by the .NET runtime, or which are considered of a very generic nature and may be thrown by almost any application.

System.ApplicationException: This class is important, because it is the intended base for any class of exception defined by users. Hence, if you define any exceptions covering error conditions unique to your application, you should derive these directly or indirectly from `System.ApplicationException`.



Overview

C# exception handling is managed via four keywords:

- try, catch, throw, and finally.
- Try block:
 - Contains program statements you wish to monitor for exceptions.
 - If an exception occurs within the try block, it is thrown.
- Catch block:
 - Your code catches this exception using catch and handles it in some rational manner.
- Finally block:
 - Any code that you must execute after you exit a try block is put in a finally block.

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.

You can explicitly generate exceptions with a program that uses 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, it allows a program to release resources.



Overview (contd..)

An exception generates when your application encounters an exceptional circumstance.

- Division by zero or low memory warning.

Flow of control immediately jumps to an associated exception handler, if one is present.

- If no handler is present, program execution stops with an error message.

Actions that result in an exception are executed with the try keyword.

In order to deal with possible error conditions in C# code, you normally divide the relevant part of your program into blocks of three different types:

- **Try blocks** contain code that forms part of the normal operation of your program, but which might encounter some serious error conditions
- **Catch blocks** contain the code that deals with the various error conditions.
- **Finally blocks** contain the code that cleans up any resources or takes any other action that you will normally want done at the end of a try or catch block. It is important to understand that the finally block is executed whether or not any exception is thrown. Since the aim is that the finally block contains cleanup code that should always be executed, the compiler will flag an error if you place a return statement inside a finally block.



```
try {  
    // Code to try here.  
}  
catch (System.Exception ex) {  
    // Code to handle exception here.  
}  
finally {  
    // Code to execute after try (and possibly catch) here.  
}
```

If the code is running in the try block, how does it know to switch to the catch block if an error has occurred? If an error is detected, the code does something that is known as **throwing an exception**. In other words, it instantiates an exception object class and throws it.

So how do these blocks fit together to trap error conditions? Here's how:

- 1.The execution flow enters a try block.
- 2.If no errors occur, execution proceeds normally through the try block, and when the end of the try block is reached, the flow of execution jumps to the finally block (Step 5). However, if an error occurs within the try block, execution jumps to a catch block (next step).
- 3.The error condition is handled in the catch block.
- 4.At the end of the catch block, execution automatically transfers to the finally block.
- 5.The finally block is executed.

Actually, there are a few variations on this theme:

- You can omit the finally block.
- You can also supply as many catch blocks as you want to handle different types of error.
- You can omit the catch blocks altogether, in which case the syntax serves not to identify exceptions, but as a way of guaranteeing that code in the finally block will be executed when execution leaves the try block. This is useful if there are several exit points in the try

block.

Demonstration: Exception Handling





Overview

- Associate more than one catch statement with a try.
- Each catch must catch a different type of exception.
- If you wish to use an Exception class in multiple catch statements, it should be the last catch statement.

In general, catch expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other catch blocks are ignored.



Commonly Used Exceptions

ArrayTypeMismatchException:

- Type of value stored is incompatible with the array type.

DivideByZeroException:

- Division by zero attempted.

IndexOutOfRangeException:

- Array index is out of bounds.

InvalidCastException:

- A runtime cast is invalid.

NullReferenceException:

- Attempt to operate on a null reference (reference that does not refer to an object).

NullReferenceException is thrown when there is an attempt to use a null reference as if it referred to an object—for example, if you attempt to call a method on a null reference. A *null reference* is a reference that does not point to any object. One way to create a null reference is to explicitly assign it the value **null** by using the keyword **null**. Null references can also occur in other ways that are less obvious.

The *System* namespace defines several standard, built-in exceptions. All are derived from *SystemException* since they are generated by the CLR when runtime errors occur. Several of the more commonly used standard exceptions defined by C# are shown in the above slide.



User Defined Exceptions

Although C#'s built-in exceptions handle most common errors, C#'s exception handling mechanism is not limited to them.

You can use custom exceptions to handle errors in your own code.

As a general rule, exceptions you define should be derived from `ApplicationException` as this is the hierarchy reserved for application-related exceptions.

Although C#'s built-in exceptions handle most common errors, C#'s exception handling mechanism is not limited to these errors. In fact, part of the power of C#'s approach to exceptions is its ability to handle exceptions that you create. You can use custom exceptions to handle errors in your own code. Creating an exception type is easy. Just define a class derived from `Exception`. As a general rule, exceptions defined by you should be derived from `ApplicationException` since this is the hierarchy reserved for application-related exceptions. Your derived classes don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The exception classes that you create will automatically have the properties and methods defined by `Exception` available to them. Of course, you can override one or more of these members in exception classes that you create.



User Defined Exceptions

```
class MyException : ApplicationException
{
    public MyException(string str):base(str)
    {
        Console.WriteLine("User defined exception");
    }
}
class MyClient
{
    public static void Main()
    {
        try
        {
            throw new MyException("Some error has
happened");
        }
        catch(MyException e)
        {
            Console.WriteLine("Exception caught here" +
e.ToString());
        }
        Console.WriteLine("LAST STATEMENT"); }}}
```

In C#, it is possible to create our own exception class. But Exception must be the ultimate base class for all exceptions in C#. So the user-defined exception classes must inherit from either Exception class or one of its standard derived classes.

Throwing an exception -

In addition to handling exceptions that are generated by .NET Framework code, you can also throw exceptions from your own code. You can use this extremely useful technique to deal with any errors that may occur. It is advantageous because you can use the powerful structured exception handling framework to transfer control from one location to another effortlessly. Without this capability, you would have to write intricate conditional code and use return values to indicate the success or failure of a method or other code.

Demonstration: User Defined Exceptions



Summary



In this module you learned:

- Need for Exception Handling
- Exception Handling using try, catch and finally block.
- Creating a User Defined Exception by inheriting it from a General Exception class.



Review Question



1. Why do you need exception Handling?
2. Can I have multiple catch blocks written for one try block?
3. What is the use of finally block?
4. How do you create an user defined exception?
5. What should be placed first, General Exception or specific Exception?





People matter, results count.

This message contains information that may be privileged or confidential and is the property of the Capgemini Group.
Copyright © 2017 Capgemini. All rights reserved.
Rightshore® is a trademark belonging to Capgemini.

About Capgemini

With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the *Collaborative Business Experience™*, and draws on *Rightshore®*, its worldwide delivery model.

Learn more about us at
www.capgemini.com

This message is intended only for the person to whom it is addressed. If you are not the intended recipient, you are not authorized to read, print, retain, copy, disseminate, distribute, or use this message or any part thereof. If you receive this message in error, please notify the sender immediately and delete all copies of this message.