Course Name	:	Dot Net Web Developer (C# Track)
Module	:	C#
Chapter	:	Exceptions

◆ Objectives

- **Exception handling**
- **Class hierarchy**
- **Exception class**
- **try/catch clause**
- **Multiple catch clauses**
- **Nested try statements**
- **throw statement**
- **finally block**

At the end of this chapter you should be able to:

- Handle error condition in C#
- Learn how exception are defined and activated
- How to throw a exception
- How to catch the exception
- How to define your own exception
- Use the finally block
- Use Checked and Unchecked keyword

◆ Overview of Errors

- **Errors are the wrongs that can make a program go wrong.**
- **An error may produce incorrect output or may terminate the execution of the program abruptly or event may cause the system to crash**
- **Errors may broadly be classified into two types:**
  - **Compile-time errors**
  - **Run-time errors**
- **All syntax errors will be detected and displayed by the C# compiler and therefore these errors are known as compile-time errors.**
- **Programs may produce wrong results due to wrong logic or may terminates due to errors such as stack overflow. It is called runtime error.**

Errors are the wrongs that can make a program go wrong. An error may produce incorrect output or may terminate the execution of the program abruptly or event may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

Errors may broadly be classified into two types:

1. Compile-time errors
2. Run-time errors

Compile-time Errors

All syntax errors will be detected and displayed by the C# compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the **.exe** file.

Most of the compile-time errors are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character by character.

The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing doubel qotes in strings
- Use of undeclared variables
- Objects are not properly initialzed
- Bad reference to objects

Run-time errors

Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminates due to errors such as stack overflow. Most common run-time errors are:

Dividing an integer by zero
Accessing an element that is out of the bounds of an array
Trying to store a value into an array of an incompatible class or type
Trying to cast an instance of a class to one of its subclasses.
Passing a parameter that is a not a valid range or value for a method.
Trying to illegally change the state of a thread
Attempting to use a negative size for an array
Using a null object reference sa a legitimate object reference to access a method or a variable.
Converting invalid string to a number
Accessing a character that is out of bounds of a string and etc.....

When such errors are encountered, C# typically generates an error message and aborts the program.

◆ Exception

- **An *exception* is a condition that is caused by run-time error in the program.**
- **When the C# interpreter encounters an error, it creates an exception object and throws.**
- **If the exception object is not caught and handled properly, the interpreter will display an error message. And will terminate the program**
- **Error handling code performs the below tasks:**
- **Find the problem(*Hit* the exception)**
- **Inform that an error has occurred (*Throw* the exception)**
- **Receive the error information (*Catch* the exception)**
- **Take corrective actions (*Handle* the exceptions)**

An ***exception*** is a condition that is caused by run-time error in the program. When the C# executer encounters an error such as dividing an integer by zero, it creates an exception object and throws (i-e., informs us that an error has occurred).

If the exception object is not caught and handled properly, the interpreter will display an error message. And will terminate the program. If we want the program to continue with the exception of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as ***exception handling***.

The purpose of exception handling mechanism is to provide a means to detect and report an “exceptional circumstance” so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

Find the problem(***Hit*** the exception)
Inform that an error has occurred(***Throw*** the exception)
Receive the error information(***Catch*** the exception)
Take corrective actions(***Handle*** the exceptions)

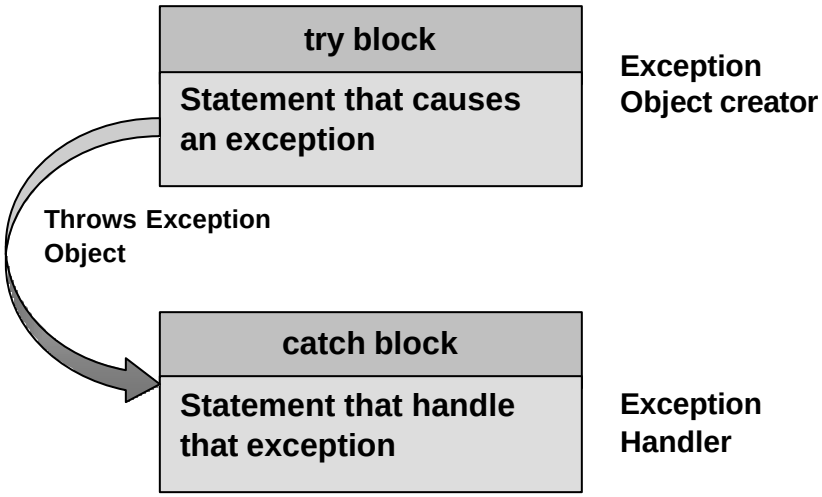
The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions. When writing programs, we must always be on the lookout for places in the program where an exception could be generated.

◆ Exception Handling Syntax

- **A C# exception is an object that describes an exceptional (I-e. error) condition that has occurred in a piece of code.**
- **When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.**
- **Either way, at some point the exception is caught and processed.**
- **Exception can be generated by the C# run-time system, or they can be manually generated by the code.**
- **C# exception handling is managed by four keywords.** ***try, catch, throw, finally***

One feature of C# language is exception handling, which allows the programmer to trap exceptions and handle them accordingly so that the program will not simply crash.

The basic concepts of exception handling are throwing an exception and catching it. This is illustrated in the below figure.



C# exception handling is managed by four keywords.

try, catch, throw, finally

Program statements that you want to monitor for exceptions are contained within a try block, it is thrown.

You code can catch this exception (using catch) and handle.

To manually throw an exception, use the keyword throw.

Any code that absolutely must be executed before a method returns is put in a finally block.

C# provides another two keywords:

checked
unchecked

C# uses the keyword **try** to preface a block of code that is likely to cause an error condition and “throw” an exception. If anyone statements generates an exception, the remaining statements in the block are skipped and the execution jumps to the catch block that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception. Remember that every try statement should be followed by at least one catch statement; otherwise compilation error will occur.

Note that the catch statement works like a method definition. The catch statement is passed a single parameter, which is reference to the exception object is thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution will terminate.

◆ Exception Handling General Form

```
try  {
    statement;          // generates exceptions
}catch(Exception-type-1 e)   {
statement;              // processes exception type 1
} catch(Exception-type-2 e)  {
statement;              // processes exception type 2
}
.
.
catch(Exception-type-N e)    {
statement;              // processes exception type N
}
finally(Exception-type-N e)  {
statement;              // executes before try block ends
}
```

It is possible to have more than one catch statement in the catch block.

```
.....................
try
{
        statement;              // generates exceptions
}catch(Exception-type-1 e)
{
        statement;              // processes exception type 1
} catch(Exception-type-2 e)
{
        statement;              // processes exception type 2
}
.
.
catch(Exception-type-N e)
{
        statement;              // processes exception type N
}
finally(Exception-type-N e)     {
        statement;              // executes before try block ends
}
```

Exception-Type denotes the type of exception that has occurred.

Note that the use of the **try … catch … finally** mechanism. This is the basic format used to trap exceptions in code. The **try… catch** portion of this operation is required syntax in C#. The **finally** keyword is used to introduce a set of statements that will always be executed before the method executes. Because exceptions are simply classes derived from the Exception class, you are free to define your own set of exception classes complete variables and methods.

When an exception in a try block is generated, the C# treats the multiple catch statements like cases in a switch statements. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

◆ Exception class

- **Only objects that are instances of this class (or of one of its subclasses) are thrown by the CLR or can be thrown by the C# throw statement**
- **Exception class is conventionally used to indicate that exceptional situations have occurred.**
- **By convention, class Exception and its subclasses have two constructors**
  Ex:
  - **ArithmeticException()**
  - **ArithmeticException(String s)**
- **The important Properties in the Exception class:**
  - **StackTrace**
  - **Message**

The Exception class is the superclass of all errors and exceptions in the C# language. Only objects that are instances of this class (or of one of its subclasses) are thrown by the Common Language Runtime(CLR) or can be thrown by the C# throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause. Instances of subclass **Exception**, are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data).

By convention, class Exception and its subclasses have two constructors, one that takes no arguments and one that takes a String argument that can be used to produce an error message. Example for the two constructors of the arithmetic class is:

1. ArithmeticException ex = new ArithmeticException();
2. ArithmeticException ex1 = new ArithmeticException(String s);

The important Properties in the Exception class:

StackTrace
Message

Exception includes a number of properties to help identify the code location and the reason for the exception. The **StackTrace** property carries a stack trace that can be used to determine where the error occurred. To improve the caller's ability to determine the reason an exception is thrown, it is desirable at times for a method to catch an exception thrown by a helper routine and, then to throw an exception more indicative of the error that has occurred. A new and more meaningful exception can be created, where the inner exception reference can be set to the original exception. This more meaningful exception can be thrown to the caller. Note that with this functionality, it is possible to

create a series of linked exceptions that terminates with the exception that was first thrown.

Exception objects also carry a message that provides details concerning the cause of an exception. This is represented by the **Message** property. To simplify the task of localization, the message strings carried by the exceptions thrown by the base class library exist in resource files.

◆ Using try…catch

- **Advantages to handle the exception yourself:**
  - **It allows handling an exception by yourself.**
  - **Prevents the program from auto terminating.**
- **Exception handling separates error from normal processing. Normal processing goes into a try block; error processing goes into a catch block.**

```
using System;  class excp   {
 static void Main()       {
try {
   int d =  0, a;
   a = 12 / d;
   Console.WriteLine("Will Not Be Printed");
 }  catch(ArithmeticException e)            {
   Console.WriteLine ("Division by zero");
} } }
```

There are two advantages to handle the exception yourself:

1. It allows to handle an exception yourself.
2. It prevents the program from automatically terminating.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch

Exception handling separates error processing from normal processing. Normal processing goes into a **try** block; error processing goes into a **catch** block.

Line 5: The arithmetic exception is raised because the 12 is divided by d where d is zero. Now the execution flow control will go to line no 7.

Line 6: This statement will not execute because the control is already passed to Line 7.

Line 7: The catch clause processes the arithmetic exception. That is, the statement inside the catch clause will be executed.

Once the catch statement is executed, program control continues with the next line in the program following the entire try/catch mechanism.

For the above program if the exception is not caught, then the following message will display by the C# runtime system and terminates the program.

Exception occurred: System.DivideByZeroException:
Attempted to divide by zero. at SimpleExceptionEx.Main()

You can have nested try and multiple catch for each try statement.

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** is statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try / catch** block.

The **try** statement can be nested. I-e, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, the C# run-time system will handle the exception.

◆ throw Statement

- **It is possible for your program to throw an exception explicitly, using the throw statement.**
- **The general form of throw is shown here:**
  - **throw throwableInstance**
  - **throwableInstance must be an object of type**
- **Exception or a subclass of Exception.**

It is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

throw throwableInstance

throwableInstance	must be an object of type **Throwable** or a subclass of **Throwable**.

The **throw** statement is executed to indicate that an exception has occurred. This is called as *throwing an exception*. Any method call can invoke code that might **throw** an exception or call another method that throws an exception.

Simple types, such as int or char as well as non-Exception classes, such as String and Object, cannot be used as exceptions. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

◆ Finally clause

- **The finally block is optional.**
- **If it is present it is placed after the last of a try block's catch blocks.**
- **C# guarantees that a finally block will be executed regardless of whether or not any exception is thrown in a try block or any of its corresponding catch block.**
- **C# also guarantees that a finally block will be executed if a try block is exited via a return, break, or continue statement.**
- **If an exception occurs, the rest of the try block is skipped and if the exception is caught by one of the catch handlers, the exception is handled and control still proceeds to the finally block.**

The finally block is optional. C# supports another statement known as **finally** statement that can be sued to handle an exception that is not caught by any of the previous catch statements. **finally** block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block.

When a **finally** is defined, this is guaranteed to execute, regardless of whether or not an exception is throws. As a result, we can use it to perform certain house-keeping operations such as closing and releasing system resources.

C# guarantees that a **finally** block will be executed regardless of whether or not any exception is thrown in a try block or any of its corresponding catch block. C# also guarantees that a **finally** block will be executed if a try block is exited via a return, break, or continue statement.

If an exception occurs, the rest of the **try** block is skipped and if the exception is caught by one of the **catch** handlers, the exception is handled and control still proceeds to the **finally** block.

If an exception that occurs in the try block cannot be caught by one of the catch handlers, the rest of the try block is skipped and control proceeds to the finally block which releases the resource and then the exception is passed up the call chain until some calling method chooses to catch it. If no method chooses to deal with it, a non-GUI based application terminates.

◆ Checked and Unchecked

- **C# statements can execute in either checked or unchecked context.**
- **In a checked context, arithmetic overflow raises an exception.**
- **In an unchecked context, arithmetic overflow is ignored and the result is truncated.**
  - **checked Specify checked context.**
  - **unchecked Specify unchecked context.**

C# statements can execute in either checked or unchecked context. In a checked context, arithmetic overflow raises an exception. In an unchecked context, arithmetic overflow is ignored and the result is truncated.

checked Specify checked context.
unchecked Specify unchecked context.

If neither **checked** nor **unchecked** is specified, the default context depends on external factors such as compiler options.

The following operations are affected by the overflow checking:

Expressions using the following predefined operators on integral types:
++ — - (unary) + - * /

Explicit numeric conversions between integral types.

The /checked compiler option lets you specify checked or unchecked context for all integer arithmetic statements that are not explicitly in the scope of a **checked** or **unchecked** keyword.

◆ Checked Keyword

- **The checked keyword is used to control the overflow-checking context for integral-type arithmetic operations and conversions.**
- **It can be used as an operator or a statement according to the following forms.**

  The checked statement:
  checked *block*

  The checked operator:
  checked (*expression*)

The **checked** keyword is used to control the overflow-checking context for integral-type arithmetic operations and conversions. It can be used as an operator or a statement according to the following forms.

The **checked** statement:
checked *block*

The **checked** operator:
checked (*expression***)**

where:
block indicates the statement block that contains the expressions to be evaluated in a checked context.
expression indicates the expression to be evaluated in a checked context. Notice that the expression must be in parentheses ( ).

In a checked context, if an expression produces a value that is outside the range of the destination type, the result depends on whether the expression is constant or non-constant. Constant expressions cause compile time errors, while non-constant expressions are evaluated at run time and raise exceptions.

If neither **checked** nor unchecked is used, a constant expression uses the default overflow checking at compile time, which is **checked**. Otherwise, if the expression is non-constant, the run-time overflow checking depends on other factors such as compiler options and environment configuration.

◆ Unchecked keyword

- **The unchecked keyword is used to control the overflow-checking context for integral-type arithmetic operations and conversions.**
- **It can be used as an operator or a statement according to the following forms.**

The unchecked statement:
unchecked *block*

The unchecked operator:
unchecked (*expression*)

The **unchecked** keyword is used to control the overflow-checking context for integral-type arithmetic operations and conversions. It can be used as an operator or a statement according to the following forms.

The **unchecked** statement:
unchecked *block*

block indicates the statement block that contains the expressions to be evaluated in an unchecked context.

The **unchecked** operator:
unchecked (*expression***)**

expression indicates the expression to be evaluated in an unchecked context. Notice that the expression must be in parentheses ( ).

In an unchecked context, if an expression produces a value that is outside the range of the destination type, the result is truncated.

If neither checked nor **unchecked** is used, a constant expression uses the default overflow checking at compile time, which is **checked**. Otherwise, if the expression is non-constant, the run-time overflow checking depends on other factors such as compiler options and environment configuration.

◆ Summary

- Some common examples of exceptions are memory exhaustion, an out-of-bounds array subscript, arithmetic overflow, division by zero, and invalid method parameters. Exception handling is designed for dealing with synchronous malfunctions, I-e., those that occur as the result of a program's execution.

- Exception handling is typically used in situations in which a malfunction will be dealt with in a different scope from that which detected the malfunction. Exception should not be used as an alternate mechanism for specifying flow of control. Exception handling should be used to process exceptions from software components such as methods, and classes that are likely to be widely used, and where it does not make sense for those components to handle their own exceptions.

- C# exception handling is geared to situations in which the method that detects an error is unable to deal with it. Such a method will throw an exception. If the exception matches the type of the parameter in one of the catch blocks, the code for that catch block is executed.

- The programmer encloses in a try block the code that may generate an error that will produce an exception. The try block is immediately followed by one or more catch blocks. Each catch block specifies the type of exception it can catch and handle. Each catch block is an exception handler.

- To manually throw an exception, we use the keyword called throw. A throws clause lists the checked exceptions that may be thrown from a method. C# guarantees that a **finally** block will be executed regardless of whether or not any exception is thrown in a try block or any of its corresponding catch block