# Exception Handling

# Lesson Objectives

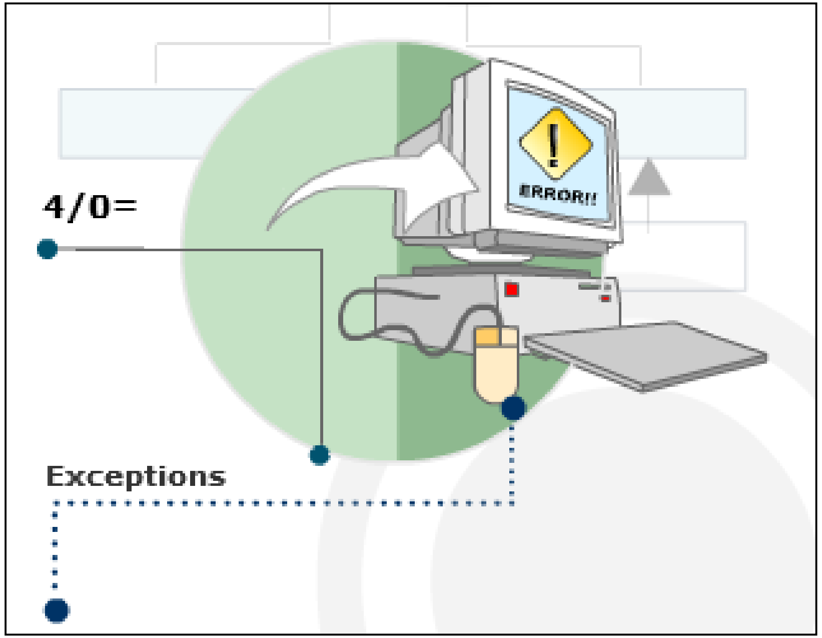
- Define and describe exceptions
- Exception vs Error
- Explain the process of throwing and catching exceptions
- Explain nested try and multiple catch blocks
- Define and describe custom exceptions
- Exception Filter

Section 1

## EXCEPTION HANDLING

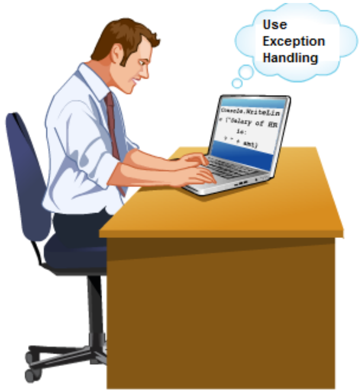
## Purpose

- Exceptions are abnormal events that prevent a certain task from being completed successfully.



# Exceptions in C#

- Consider a C# application that is currently being executed.
- Assume that at some point of time, the CLR discovers that it does not have the read permission to read a particular file.
- The CLR immediately stops further processing of the program and terminates its execution abruptly.
- To avoid this from happening, you can use the exception handling features of C#.

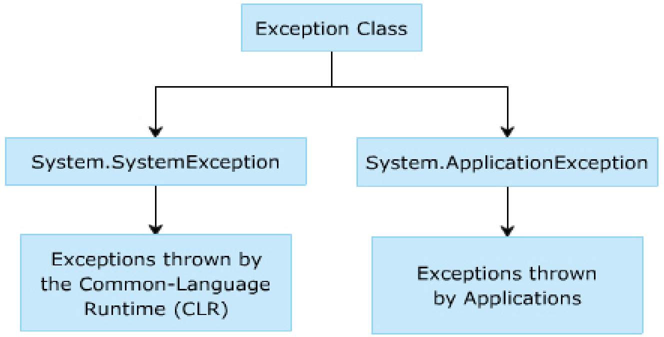


# Types of Exceptions in C#

- C# can handle different types of exceptions using exception handling statements.
- It allows you to handle basically two kinds of exceptions. These are as follows:
  - ✓ **System-level Exceptions**: These are the exceptions thrown by the system that are thrown by the CLR.

    For example, exceptions thrown due to failure in database connection or network connection are system-level exceptions.
  - ✓ **Application-level Exceptions**: These are thrown by user-created applications.

    For example, exceptions thrown due to arithmetic operations or referencing any null object are application-level exceptions.
- The following figure displays the types of exceptions in C#:


Exception Class
System.SystemException
System.ApplicationException
Exceptions thrown by the Common-Language Runtime (CLR)
Exceptions thrown by Applications

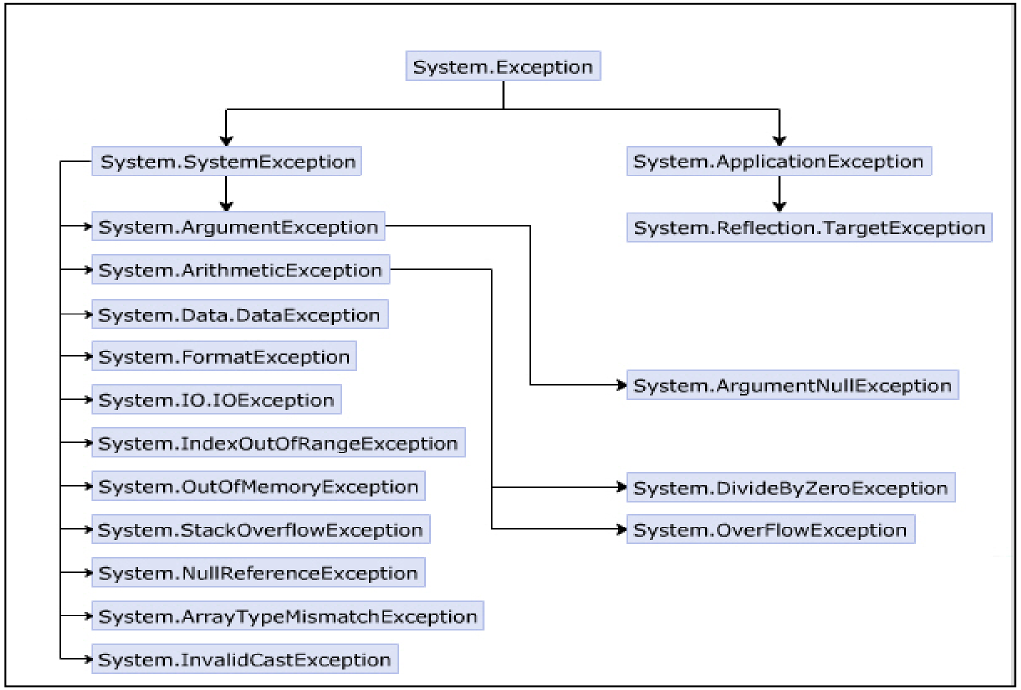
# Properties of the Exception Class

- System.Exception:
  - ✓ Is the base class that allows you to handle all exceptions in C#.
  - ✓ Is inherited by all exceptions in C# either directly or indirectly.
  - ✓ Contains public and protected methods that can be inherited by other exception classes and also contains properties that are common to all exceptions.
- The following table describes the properties of Exception class:

| **Properties** | **Descriptions** |
|---|---|
| `Message` | Displays a message which indicates the reason for the exception. |
| `Source` | Provides the name of the application or the object that caused the exception. |
| `StackTrace` | Provides exception details on the stack at the time the exception was thrown. |
| `InnerException` | Returns the `Exception` instance that caused the current exception. |

# Commonly Used Exception Classes

- The System.Exception class has a number of derived exception classes to handle different types of exceptions.
- The hierarchy shown in the following figure displays the different exception classes in the System namespace:



System.Exception
System.SystemException
System.ApplicationException
System.ArgumentException
System.Reflection.TargetException
System.ArithmeticException
System.Data.DataException
System.FormatException
System.ArgumentNullException
System.IO.IOException
System.IndexOutOfRangeException
System.OutOfMemoryException
System.DivideByZeroException
System.StackOverflowException
System.OverFlowException
System.NullReferenceException
System.ArrayTypeMismatchException
System.InvalidCastException

# Exception Classes 1-2

- The System namespace contains the different exception classes that C# provides the type of exception to be handled depends on the specified exception class.
- The following table lists some of the commonly used exception classes:

| Exceptions | Descriptions |
|---|---|
| `System.ArithmeticException` | This exception is thrown for problems that occur due to arithmetic or casting and conversion operations. |
| `System.ArgumentException` | This exception is thrown when one of the arguments does not match the parameter specifications of the invoked method. |
| `System.ArrayTypeMismatchException` | This exception is thrown when an attempt is made to store data in an array whose type is incompatible with the type of the array. |
| `System.DivideByZeroException` | This exception is thrown when an attempt is made to divide a numeric value by zero. |
| `System.IndexOutOfRangeException` | This exception is thrown when an attempt is made to store data in an array using an index that is less than zero or outside the upper bound of the array. |
| `System.InvalidCastException` | This exception is thrown when an explicit conversion from the base type or interface type to another type fails. |
| `System.ArgumentNullException` | This exception is thrown when a null reference is passed to an argument of a method that does not accept `null` values. |

# Exception Classes 2-2

- The following table lists additional exception classes:

| Exceptions | Descriptions |
|---|---|
| `System.NullReferenceException` | This exception is thrown when you try to assign a value to a `null` object. |
| `System.OutOfMemoryException` | This exception is thrown when there is not enough memory to allocate to an object. |
| `System.OverflowException` | This exception is thrown when the result of an arithmetic, casting, or conversion operation is too large to be stored in the destination object or variable. |
| `System.StackOverflowException` | This exception is thrown when the stack runs out of space due to having too many pending method calls. |
| `System.Data.DataException` | This exception is thrown when errors are generated while using the ADO.NET components. |
| `System.FormatException` | This exception is thrown when the format of an argument does not match the format of the parameter data type of the invoked method. |
| `System.IO.IOException` | This exception is thrown when any I/O error occurs while accessing information using streams, files, and directories. |

## Catching Exceptions 1-3

- Exception handling is implemented using the try-catch construct in C# that consists of the following two blocks:

**The `try` block**

- It encloses statements that might generate exceptions.
- When these exceptions are thrown, the required actions are performed using the `catch` block.

**The `catch` block**

- It consists of the appropriate error-handlers that handle exceptions.
- If the `catch` block does not contain any parameter, it can `catch` any type of exception.
- The `catch` block follows the `try` block and may or may not contain parameters.
- If the `catch` block contains a parameter, it catches the type of exception specified by the parameter.

# Catching Exceptions 2-3

- The following syntax is used for handling errors using the try and catch blocks:

**Syntax**

```
try
{
// program code
}
catch[(<ExceptionClass><objException>)]
{
// error handling code
}
```

- where,
  - ✓ try: Specifies that the block encloses statements that may throw exceptions.
  - ✓ program code: Are statements that may generate exceptions.
  - ✓ catch: Specifies that the block encloses statements that catch exceptions and carry out the appropriate actions.
  - ✓ ExceptionClass: Is the name of exception class. It is optional.
  - ✓ objException: Is an instance of the particular exception class. It can be omitted if the ExceptionClass is omitted.

## Catching Exceptions 3-3

- The code demonstrates the use of try-catch blocks.

Snippet

```
using System;
class DivisionError
{
        static void Main(string[] args)
        {
            int numOne = 133;
            int numTwo = 0;
            int result;
        try
        {
            result = numOne / numTwo;
        }
        catch (DivideByZeroException objDivide)
        {
            Console.WriteLine("Exception caught: " + objDivide);
        }
        }
}
```

# General catch Block 1-2

- Following are the features of a general catch block:

It can handle all types of exceptions.

However, the type of exception that the catch block handles depends on the specified exception class.

You can create a catch block with the base class Exception that are referred to as general catch blocks.

A general catch block can handle all types of exceptions.

However, one disadvantage of the general catch block is that there is no instance of the exception and thus, you cannot know what appropriate action must be performed for handling the exception.

## General catch Block 2-2

- The following code demonstrates the way in which a general catch block is declared:

Snippet

```
using System;
class Students
{
        string[] _names = { "James", "John", "Alexander" };
        static void Main(string[] args)
        {
        Students objStudents = new Students();
        try
        {
        objStudents._names[4] = "Michael";
        }
        catch (Exception objException)
        {
        Console.WriteLine("Error: " + objException);
        }
        }
}
```

## The throw Statement 1-2

- The throw statement in C# allows you to programmatically throw exceptions using the throw keyword.
- When you throw an exception using the throw keyword, the exception is handled by the catch block as shown in the following syntax:

Syntax

```
throw exceptionObject;
```

- where,
  - ✓ throw: Specifies that the exception is thrown programmatically.
  - ✓ exceptionObject: Is an instance of a particular exception class.

## The throw Statement 2-2

- The following code demonstrates the use of the throw statement:

Snippet

```
using System;
class Employee
{
          static void ThrowException(string name)
          {
          if (name == null)
          {
          throw new ArgumentNullException();
          }
          }
          static void Main(string [] args)
          {
               Console.WriteLine("Throw Example");
               try
               {
                   string empName = null;
                   ThrowException(empName);
               }
               catch (ArgumentNullException objNull)
               {
                   Console.WriteLine("Exception caught: " + objNull);
               }
          }
}
```

Output

```
Throw Example
Exception caught: System.ArgumentNullException: Value cannot be null.
  at Exception_Handling.Employee.ThrowException(String name) in D:\Exception Handling\Employee.cs:
line 13
 at Exception_Handling.Employee.Main(String[] args) in D:\Exception Handling\Employee.cs:line 24
```

## The `finally` Statement 1-2

- In general, if any of the statements in the try block raises an exception, the catch block is executed and the rest of the statements in the try block are ignored.
- Sometimes, it becomes mandatory to execute some statements irrespective of whether an exception is raised or not. In such cases, a finally block is used.
- The finally block is an optional block and it appears after the catch block. It encloses statements that are executed regardless of whether an exception occurs.
- The following syntax demonstrates the use of the finally block:

Syntax

```
finally
{

// cleanup code;
}
```

- where,
  - ✓ finally: Specifies that the statements in the block have to be executed irrespective of whether or not an exception is raised.

# The `finally` Statement 2-2

- The following code demonstrates the use of the try-catch-finally construct:

Snippet

```
using System;
class DivisionError
{
      static void Main(string[] args)
      {
      int numOne = 133;
      int numTwo = 0;
      int result;
      try
      {
            result = numOne / numTwo;
      }
      catch (DivideByZeroException objDivide)
      {
           Console.WriteLine("Exception caught: " + objDivide);
      }
      finally
      {
           Console.WriteLine("This finally block will always be
           executed");
      }
      }

}
```

Output

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at DivisionError.Main(String[] args)

This finally block will always be executed
```

# Nested `try` and Multiple `catch` Blocks

- Exception handling code can be nested in a program. In nested exception handling, a try block can enclose another try-catch block.
- In addition, a single try block can have multiple catch blocks that are sequenced to catch and handle different type of exceptions raised in the try block.

# Nested `try` Blocks 1-2

- Following are the features of the nested try block:

The nested try block consists of multiple `try-catch` constructs that starts with a `try` block, which is called the outer `try` block.

This outer `try` block contains multiple `try` blocks within it, which are called inner `try` blocks.

If an exception is thrown by a nested `try` block, the control passes to its corresponding nested `catch` block.

Consider an outer `try` block containing a nested `try-catch-finally` construct. If the inner `try` block throws an exception, control is passed to the inner `catch` block.

However, if the inner `catch` block does not contain the appropriate error handler, the control is passed to the outer `catch` block.

# Nested `try` Blocks 2-2

- The following syntax is used to create nested try…catch blocks:

Syntax

```
try
{
// outer try block
try
{
// inner try block
}
catch
{
// inner catch block
}
// this is optional
finally
{
// inner finally block
}
}
catch
{
// outer catch block
}
// this is optional
finally
{
```

## Multiple catch Blocks

- A try block can throw multiple types of exceptions, which need to be handled by the catch block. C# allows you to define multiple catch blocks to handle the different types of exceptions that might be raised by the try block. Depending on the type of exception thrown by the try block, the appropriate catch block (if present) is executed.
- However, if the compiler does not find the appropriate catch block, then the general catch block is executed.
- Once the catch block is executed, the program control is passed to the finally block (if any) and then the control terminates the try-catch-finally construct.
- The following syntax is used for defining multiple catch blocks:

```
try
{
// program code
}
catch (<ExceptionClass><objException>)
{
// statements for handling the exception
}
catch (<ExceptionClass1><objException>)
{
// statements for handling the exception
}
```

# Exception Handling - Summary

- Exceptions are errors that encountered at run-time.
- Exception-handling allows you to handle methods that are expected to generate exceptions.
- The try block should enclose statements that may generate exceptions while the catch block should catch these exceptions.
- The finally block is meant to enclose statements that need to be executed irrespective of whether or not an exception is thrown by the try block.
- Nested try blocks allow you to have a try-catch-finally construct within a try block.
- Multiple catch blocks can be implemented when a try block throws multiple types of exceptions.

Section 2

## CUSTOM EXCEPTIONS

## Custom Exceptions

- Following are the features of custom exceptions:

They are user-defined exceptions that allow users to recognize the cause of unexpected events in specific programs and display custom messages.

Allows you to simplify and improve the process of error-handling in a program.

Even though C# provides you with a rich set of exception classes, they do not address certain application-specific and system-specific constraints. To handle such constraints, you can define custom exceptions.

Custom exceptions can be created to handle exceptions that are thrown by the `CLR` as well as those that are thrown by user applications.

## Implementing Custom Exceptions

- Custom exceptions can be created by deriving from the Exception class, the SystemException class, or the ApplicationException class.
- However, to define a custom exception, you first need to define a new class and then derive it from the Exception, SystemException, or ApplicationException class. Once you have created a custom exception, you can reuse it in any C# application.
- The following demonstrates the implementation of custom exceptions:

```
public class CustomMessage : Exception
{
public CustomMessage (string message) : base(message) {
}
}
public class CustomException{
static void Main(string[] args) {
try
{
throw new CustomMessage (“This illustrates creation and catching of custom exception”);
}
catch(CustomMessage objCustom)
{
Console.WriteLine(objCustom.Message);
}
}
}
```

# User-defined Exceptions

```
using System;
class MyException : Exception
{
    public MyException(string str)
    {
        Console.WriteLine("User defined exception");
    }
}
class MyClient
{
    public static void Main()
    {
        int x = 0;
        int div = 0;

        try
        {
            div = 100 / x;
            throw new MyException("rajesh");
        }
        catch (Exception 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.

## Custom Exception - Summary

- Custom exceptions are user-defined exceptions that allow users to handle system and application-specific runtime errors.

Section 3

## EXCEPTION HANDLING GUIDELINES

# Exception Handling Guidelines 1/6

- Exceptions should be used to communicate exceptional conditions.
- Don't use them to communicate events that are expected, such as reaching the end of a file.
- If there's a good predefined exception in the System namespace that describes the exception condition (one that will make sense to the users of the class) use that one rather than defining a new exception class, and put specific information in the message.

# Exception Handling Guidelines 2/6

- *Do Not Catch Exceptions That You Cannot Handle*.

```
try { intNumber = int.Parse(strNumber);
      }
  catch (Exception ex)
 {    Console.WriteLine("Can't convert the string to " + "a number: " + ex.Message);}
```

```
try { intNumber = int.Parse(strNumber);
      }
 catch (ArgumentNullException ex)
     { Console.WriteLine(@"input is null"); }
  catch( FormatException   ex)
    { Console.WriteLine(@"Incorrect format"); }
```

✓You should never catch System.Exception or System.SystemException in a catch block because you could inadvertently hide run-time problems like Out Of Memory.

# Exception Handling Guidelines 3/6

- Use validation code to avoid unnecessary exceptions.

```
double result = 0;
 try{
 result = numerator/divisor; }
 catch( System.Exception e)
{ result = System.Double.NaN; }
```

more efficient.

```
double result = 0;
if ( divisor != 0 ) result = numerator/divisor;
 else result = System.Double.NaN;
```

# Exception Handling Guidelines 4/6

- Do Not Use Exceptions to Control Application Flow

```
static void ProductExists( string ProductId)
{ //... search for Product
if ( dr.Read(ProductId) ==0 )  // no record found, ask to create { throw( new Exception("Product Not found"));
} }
```

```
static bool ProductExists( string ProductId)
 { //... search for Product
 if ( dr.Read(ProductId) ==0 ) // no record found, ask to create { return false; } . . .
}
```

# Exception Handling Guidelines 5/6

- The following code ensures that the connection is always closed.

```
SqlConnection conn = new SqlConnection("...");
try { conn.Open();
//.Do some operation that might cause an exception
// Calling Close as early as possible conn.Close();
 // ... other potentially long operations
 } Finally
 { if (conn.State==ConnectionState.Open) conn.Close(); // ensure that the connection is closed }
```

# Exception Handling Guidelines 6/6

- The cost of using throw to rethrow an existing exception is approximately the same as throwing a new exception. In the following code, there is no savings from rethrowing the existing exception.

```
try { // do something that may throw an exception } catch (Exception e) { // do something with e throw; }
```

- Do not catch exceptions that you do not know how to handle and then fail to propagate the exception

```
try {
 // exception generating code
} catch(Exception e)  {
 // Do nothing
 }
```

## Exception Handling Guidelines - Summary

- Exceptions should be used to communicate exceptional conditions
- Do Not Catch Exceptions That You Cannot Handle
- Use validation code to avoid unnecessary exceptions
- Do Not Use Exceptions to Control Application Flow
- The following code ensures that the connection is always closed.

## Lesson Summary

- Exceptions are errors that encountered at run-time.
- Exception-handling allows you to handle methods that are expected to generate exceptions.
- The try block should enclose statements that may generate exceptions while the catch block should catch these exceptions.
- The finally block is meant to enclose statements that need to be executed irrespective of whether or not an exception is thrown by the try block.
- Nested try blocks allow you to have a try-catch-finally construct within a try block.
- Multiple catch blocks can be implemented when a try block throws multiple types of exceptions.
- Custom exceptions are user-defined exceptions that allow users to handle system and application-specific runtime errors
- Exception Handling Guidelines

# Thank you