

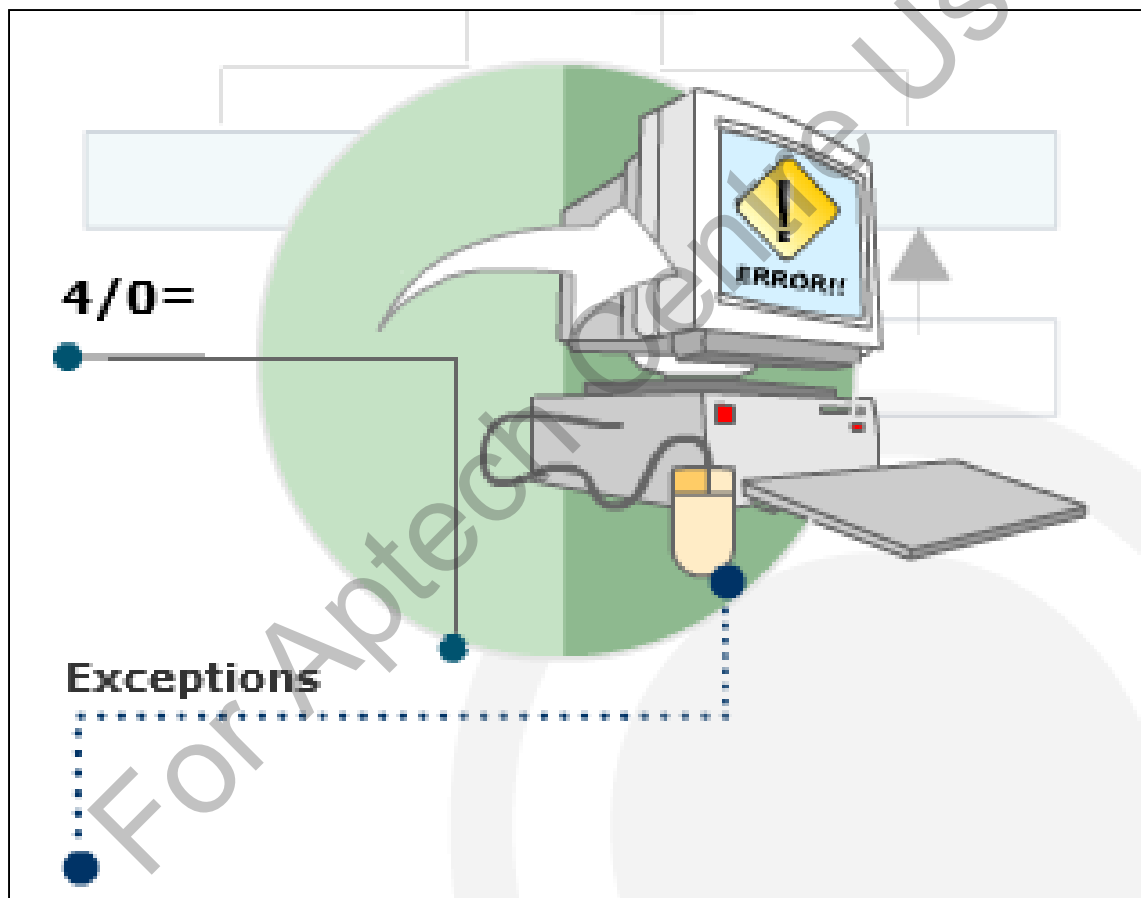
Session: **11**

# Exception Handling

- ◆ Define and describe exceptions
- ◆ Explain the process of throwing and catching exceptions
- ◆ Explain nested `try` and multiple `catch` blocks
- ◆ Define and describe custom exceptions

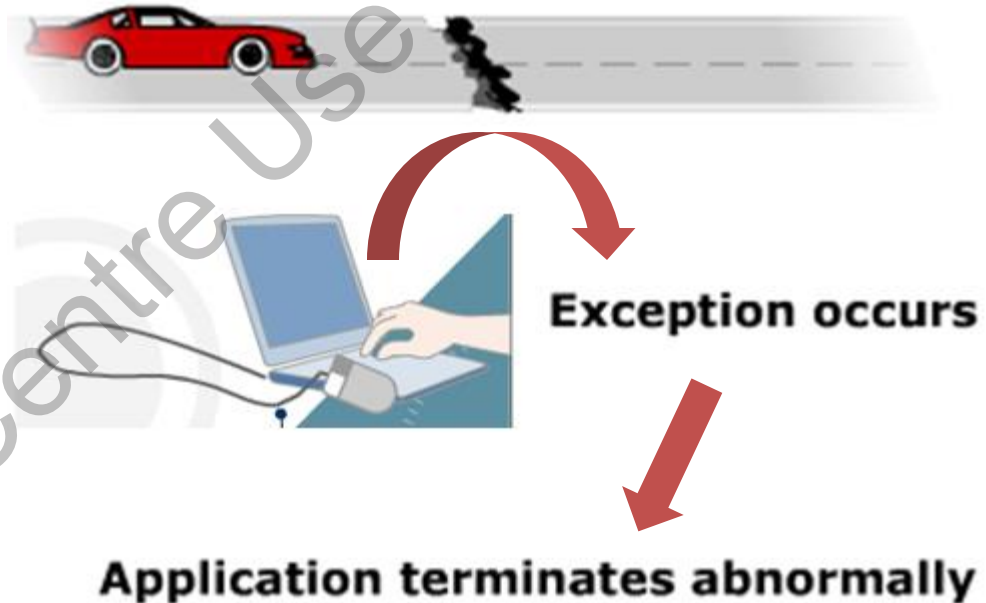
For Aptech Centre Use Only

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



### Example

- ◆ Consider a vehicle that halts abruptly due to some problem in the engine.
- ◆ Until this problem is sorted out, the vehicle may not move ahead.
- ◆ Similarly, in C#, exceptions disrupt the normal flow of the program.

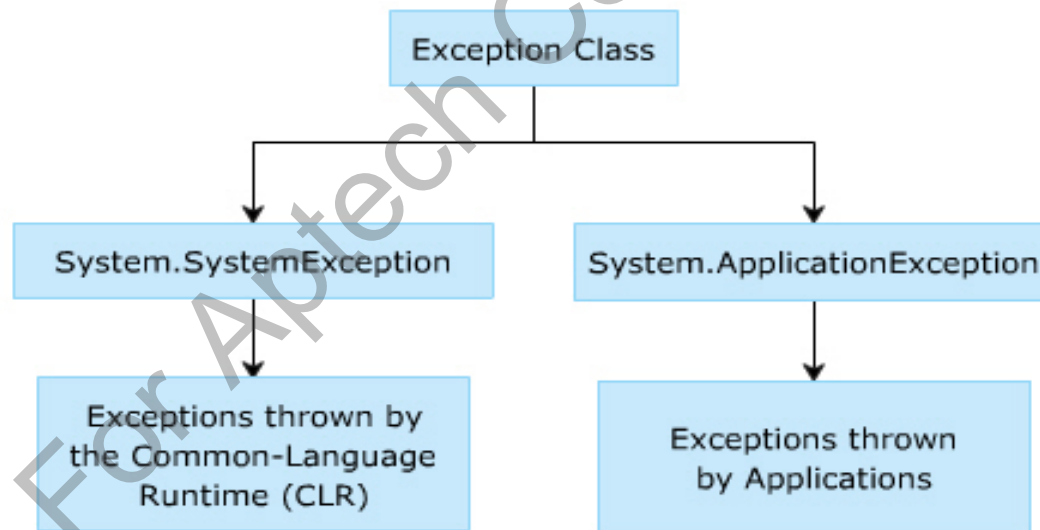


- ◆ Consider a C# application that is currently being executed.
- ◆ Assume that at some point of time, the CLR discovers that it does not have the read permission to read a particular file.
- ◆ The CLR immediately stops further processing of the program and terminates its execution abruptly.
- ◆ To avoid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



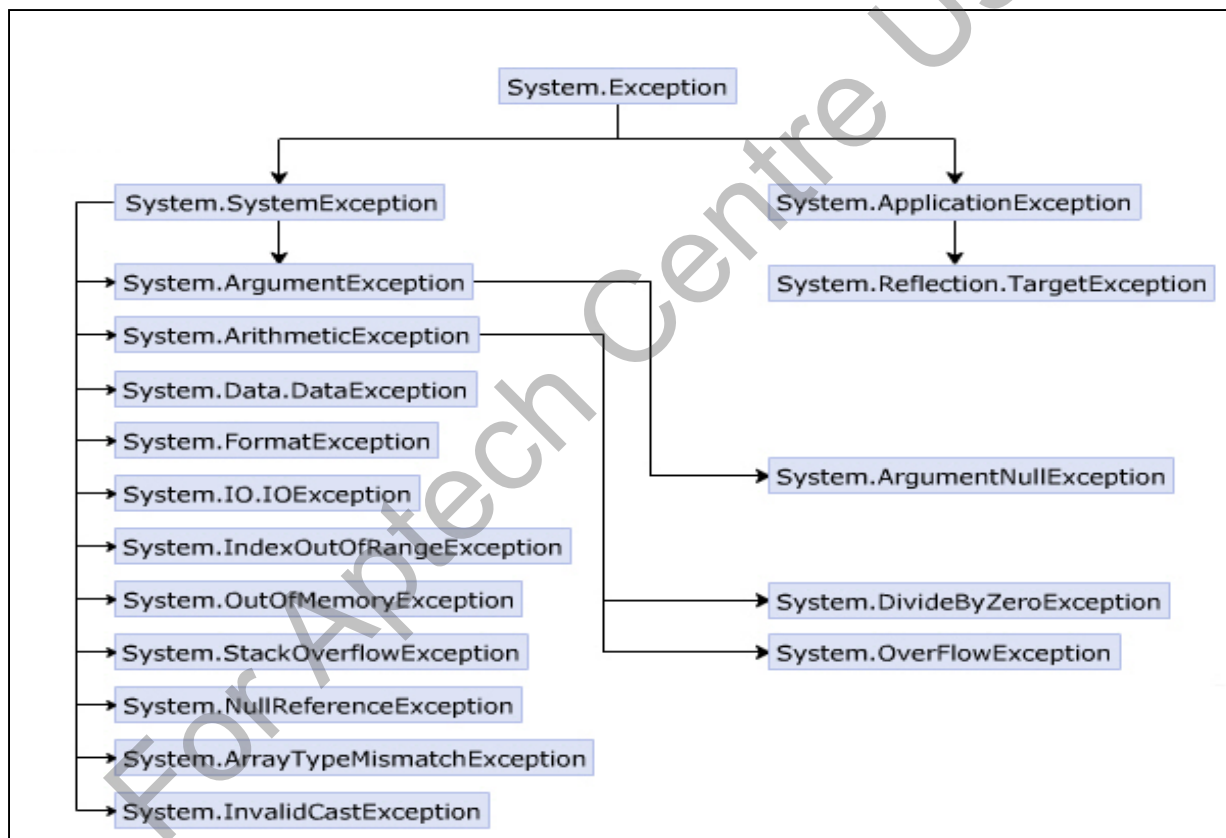
- ◆ `System.Exception`:
  - ◆ Is the base class that allows you to handle all exceptions in C#.
  - ◆ Is inherited by all exceptions in C# either directly or indirectly.
  - ◆ Contains public and protected methods that can be inherited by other exception classes and also contains properties that are 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 <code>Exception</code> 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:





## 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
<code>System.ArithmeticException</code>	This exception is thrown for problems that occur due to arithmetic or casting and conversion operations.
<code>System.ArgumentException</code>	This exception is thrown when one of the arguments does not match the parameter specifications of the invoked method.
<code>System.ArrayTypeMismatchException</code>	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.
<code>System.DivideByZeroException</code>	This exception is thrown when an attempt is made to divide a numeric value by zero.
<code>System.IndexOutOfRangeException</code>	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.
<code>System.InvalidCastException</code>	This exception is thrown when an explicit conversion from the base type or interface type to another type fails.
<code>System.ArgumentNullException</code>	This exception is thrown when a null reference is passed to an argument of a method that does not accept <code>null</code> values.

- ◆ The following table lists additional exception classes:

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

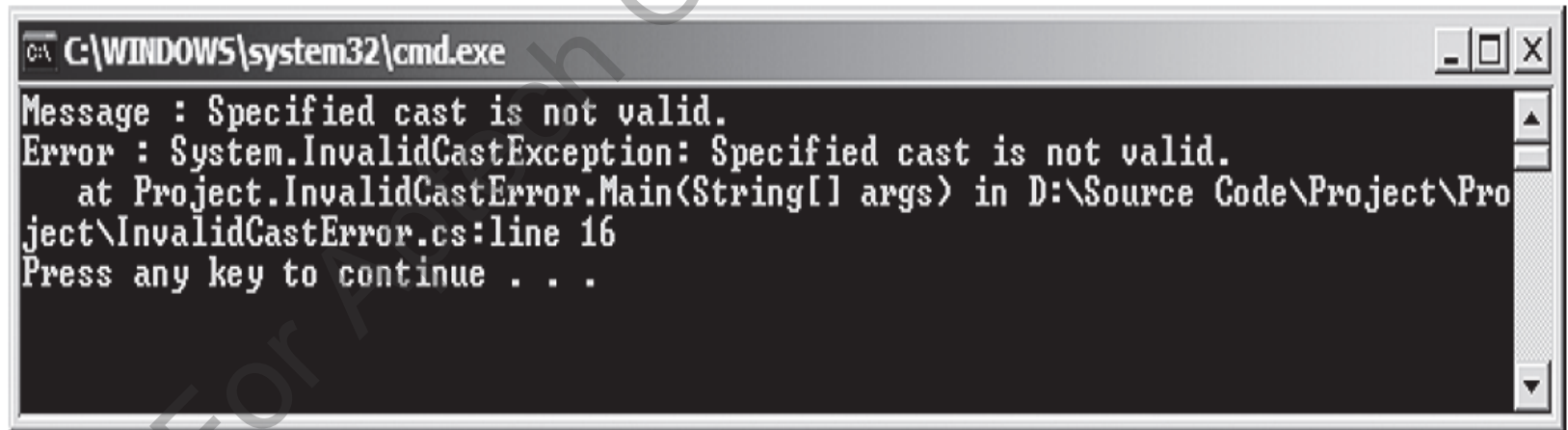
## InvalidCastException Class 1-2

- ◆ The `InvalidCastException` exception is thrown when an explicit conversion from a base type to another type fails.
- ◆ The following code demonstrates the `InvalidCastException` exception:

### Snippet

```
using System;
class InvalidCastError
{
    static void Main(string[] args)
    {
        try
        {
            float numOne = 3.14F;
            Object obj = numOne;
            int result = (int)obj;
            Console.WriteLine("Value of numOne = {0}", result);
        }
        catch(InvalidCastException objEx)
        {
            Console.WriteLine("Message : {0}", objEx.Message);
            Console.WriteLine("Error : {0}", objEx);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error : {0}", objEx);
        }
    }
}
```

- ◆ In the code:
  - ◆ A variable `numOne` is defined as `float`. When this variable is boxed, it is converted to type `object`.
  - ◆ However, when it is unboxed, it causes an `InvalidCastException` and displays a message for the same.
  - ◆ This is because a value of type `float` is unboxed to type `int`, which is not allowed in C#.
- ◆ The figure displays the exception that is generated when the program is executed.



```
C:\WINDOWS\system32\cmd.exe
Message : Specified cast is not valid.
Error : System.InvalidCastException: Specified cast is not valid.
       at Project.InvalidCastError.Main(String[] args) in D:\Source Code\Project\Pro
ject\InvalidCastError.cs:line 16
Press any key to continue . . .
```

# ArrayTypeMismatchException Class 1-2

- ◆ The `ArrayTypeMismatchException` exception is thrown when the data type of the value being stored is incompatible with the data type of the array.
- ◆ The following code demonstrates the `ArrayTypeMismatchException` exception:

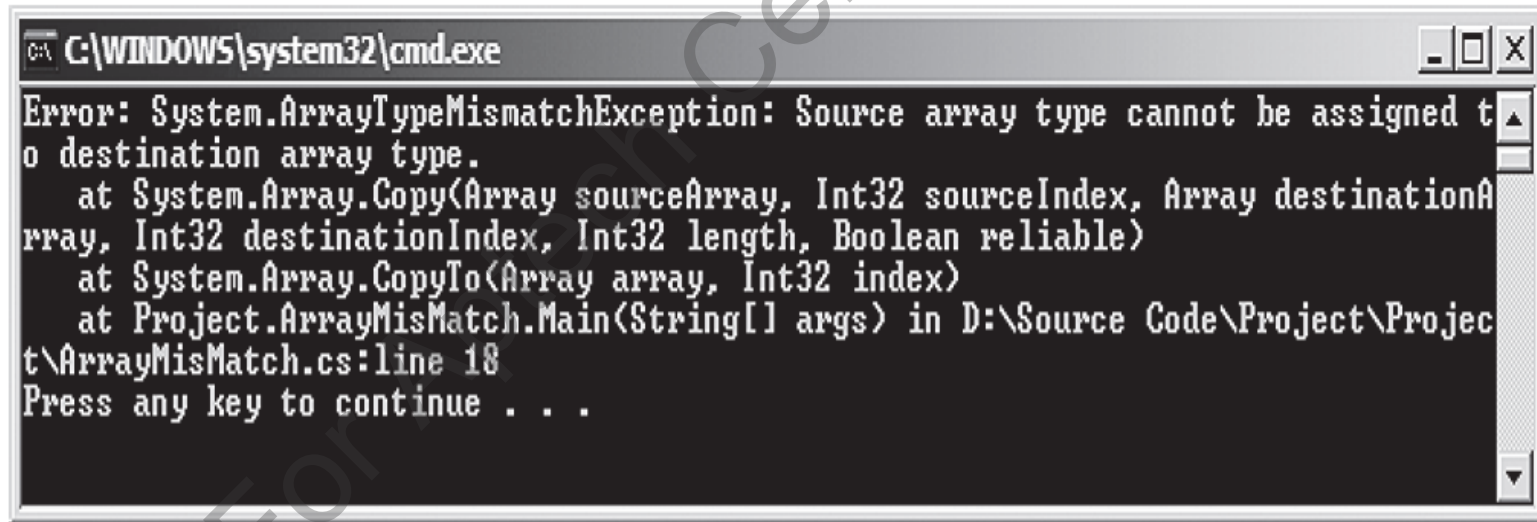
## Snippet

```
using System;
class ArrayMisMatch
{
    static void Main(string[] args)
    {
        string[] names = { "James", "Jack", "Peter" };
        int[] id = { 10, 11, 12 };
        double[] salary = { 1000, 2000, 3000 };
        float[] bonus = new float[3];

        try
        {
            salary.CopyTo(bonus, 0);
        }
        catch (ArrayTypeMismatchException objType)
        {
            Console.WriteLine("Error: " + objType);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error: " + objEx);
        }
    }
}
```

## ArrayTypeMismatchException Class 2-2

- ◆ In the code:
  - ◆ The **salary** array is of double data type and the **bonus** array is of float data type.
  - ◆ When copying the values stored in **salary** array to **bonus** array using the `CopyTo()` method, data type mismatch occurs.
- ◆ The following table displays the exception that is generated when the program is executed:



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window displays the following error message:

```
Error: System.ArrayTypeMismatchException: Source array type cannot be assigned to destination array type.  
    at System.Array.Copy(Array sourceArray, Int32 sourceIndex, Array destinationArray, Int32 destinationIndex, Int32 length, Boolean reliable)  
    at System.Array.CopyTo(Array array, Int32 index)  
    at Project.ArrayMisMatch.Main(String[] args) in D:\Source Code\Project\Project\ArrayMisMatch.cs:line 18  
Press any key to continue . . .
```



# NullReferenceException Class 1-2

- ◆ The `NullReferenceException` exception is thrown when an attempt is made to operate on a null reference as shown in the following code:

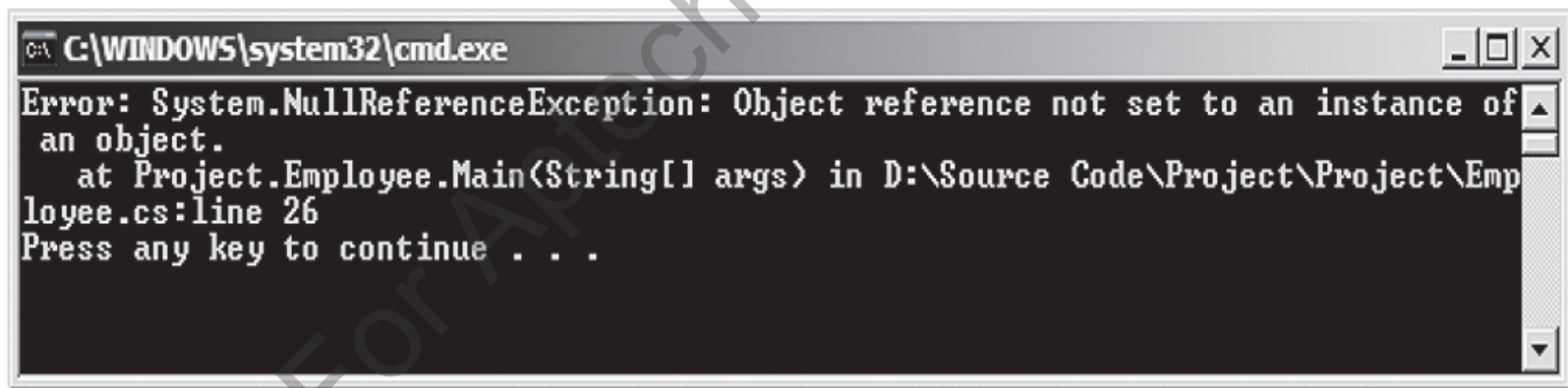
## Snippet

```
using System;
class Employee
{
    private string _empName;
    private int _empID;
    public Employee()
    {
        _empName = "David";
        _empID = 101;
    }
    static void Main(string[] args)
    {
        Employee objEmployee = new Employee();
        Employee objEmp = objEmployee;
        objEmployee = null;
        try
        {
            Console.WriteLine("Employee Name: " +
                objEmployee._empName);
            Console.WriteLine("Employee ID: " +
                objEmployee._empID);
        }
        catch (NullReferenceException objNull)
        {
            Console.WriteLine("Error: " + objNull);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error: " + objEx);
        }
    }
}
```



## NullReferenceException Class 2-2

- ◆ In the code:
  - ◆ A class **Employee** is defined which contains the details of an employee. Two instances of class **Employee** are created with the second instance referencing the first.
  - ◆ The first instance is de-referenced using null. If the first instance is used to print the values of the **Employee** class, a `NullReferenceException` exception is thrown, which is caught by the catch block.
- ◆ The figure displays the exception that is generated when the program is executed.



```
C:\WINDOWS\system32\cmd.exe
Error: System.NullReferenceException: Object reference not set to an instance of
an object.
   at Project.Employee.Main(String[] args) in D:\Source Code\Project\Project\Emp
loyee.cs:line 26
Press any key to continue . . .
```

## System.Exception Class 1-5

- ◆ `System.Exception` is the base class that handles all exceptions and contains `public` and `protected` methods that can be inherited by other exception classes.
- ◆ The following table lists the `public` methods of the `System.Exception` class and their corresponding description:

Method	Description
<code>Equals</code>	Determines whether objects are equal
<code>GetBaseException</code>	Returns a type of Exception class when overridden in a derived class
<code>GetHashCode</code>	Returns a hash function for a particular type
<code>GetObjectData</code>	Stores information about serializing or deserializing a particular object with information about the exception when overridden in the inheriting class
<code>GetType</code>	Retrieves the type of the current instance
<code>ToString</code>	Returns a string representation of the thrown exception

## System.Exception Class 2-5

- ◆ The following code demonstrates some public methods of the System.Exception class:

### Snippet

```
using System;
class ExceptionMethods
{
    static void Main(string[] args)
    {
        byte numOne = 200;
        byte numTwo = 5;
        byte result = 0;
        try
        {
            result = checked((byte)(numOne * numTwo));
            Console.WriteLine("Result = {0}", result);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error Description : {0}",
                objEx.ToString());
            Console.WriteLine("Exception : {0}", objEx.GetType());
        }
    }
}
```

- ◆ In the code:
  - ◆ The arithmetic overflow occurs because the **result** of multiplication of two byte numbers exceeds the range of the data type of the destination variable, result.
  - ◆ The arithmetic overflow exception is thrown and it is caught by the `catch` block. The block uses the `ToString()` method to retrieve the string representation of the exception.
  - ◆ The `GetType()` method of the `System.Exception` class retrieves the type of exception which is then displayed by using the `WriteLine()` method.

## System.Exception Class 3-5

- ◆ The following table lists the protected methods of the class and their corresponding descriptions:

Name	Description
Finalize	Allows objects to perform cleanup operations before they are reclaimed by the garbage collector
MemberwiseClone	Creates a copy of the current object

- ◆ In the following table lists the commonly used public properties in the class:

Public Property	Description
HelpLink	Retrieves or sets a link to the help file associated with the exception
InnerException	Retrieves the reference of the object of type Exception that caused the thrown exception
Message	Retrieves the description of the current exception
Source	Retrieves or sets the name of the application or the object that resulted in the error
StackTrace	Retrieves a string of the frames on the stack when an exception is thrown
TargetSite	Retrieves the method that throws the exception

## System.Exception Class 4-5

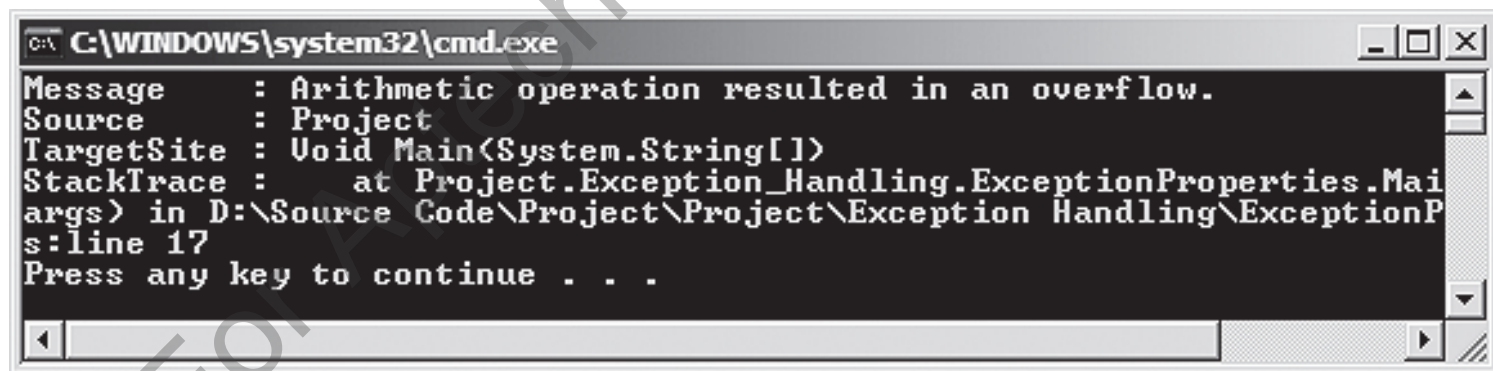
- ◆ The code demonstrates the use of some of the public properties of the System.Exception class.

### Snippet

```
using System;
class ExceptionProperties
{
    static void Main(string[] args)
    {
        byte numOne = 200;
        byte numTwo = 5;

        byte result = 0;
        try
        {
            result = checked((byte)(numOne * numTwo));
            Console.WriteLine("Result = {0}", result);
        }
        catch (OverflowException objEx)
        {
            Console.WriteLine("Message : {0}",
                objEx.Message);
            Console.WriteLine("Source : {0}",
                objEx.Source);
            Console.WriteLine("TargetSite : {0}",
                objEx.TargetSite);
            Console.WriteLine("StackTrace : {0}",
                objEx.StackTrace);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error : {0}", objEx);
        }
    }
}
```

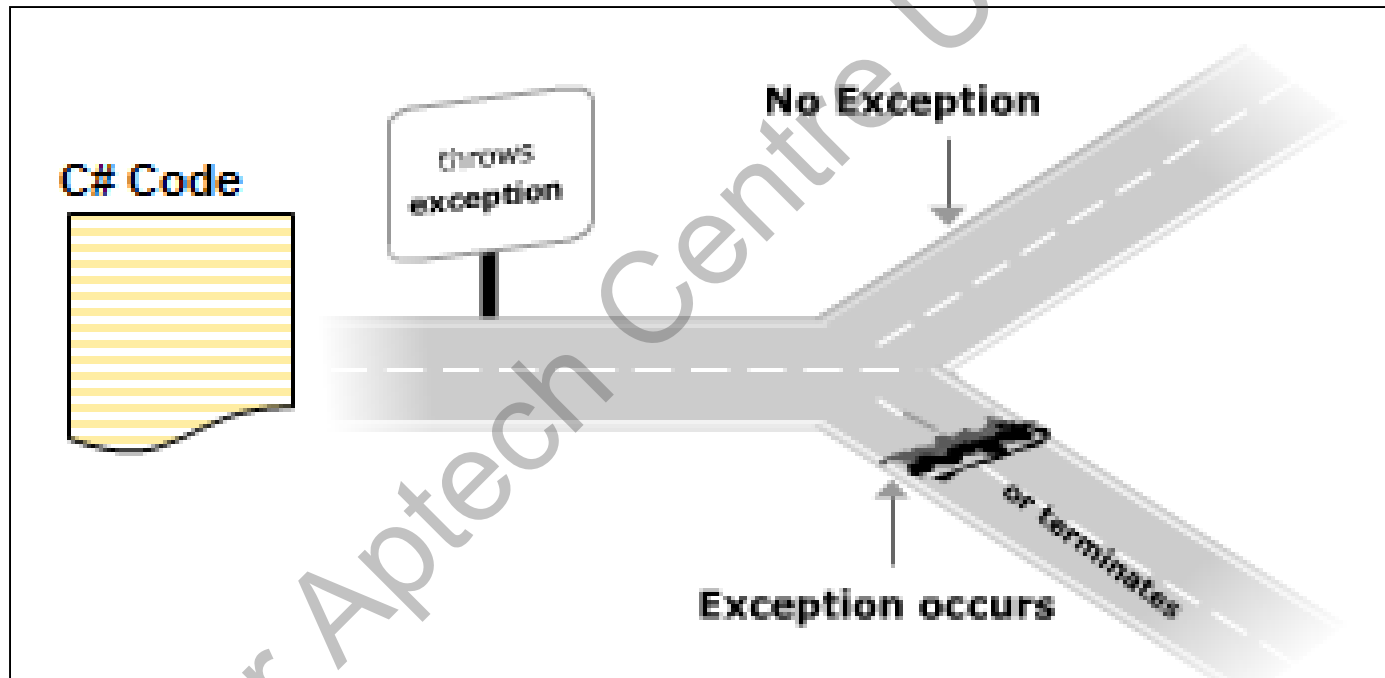
- ◆ In the code:
  - ◆ The arithmetic overflow occurs because the result of multiplication of two byte numbers exceeds the range of the destination variable type, **result**.
  - ◆ The arithmetic overflow exception is thrown and it is caught by the `catch` block.
  - ◆ The block uses various properties of the `System.Exception` class to display the source and target site of the error.
- ◆ The following figure displays use of some of the public properties of the `System.Exception` class:



```
C:\WINDOWS\system32\cmd.exe
Message      : Arithmetic operation resulted in an overflow.
Source       : Project
TargetSite   : Void Main(System.String[])
StackTrace   :    at Project.Exception_Handling.ExceptionProperties.Main
args) in D:\Source Code\Project\Project\Exception Handling\ExceptionP
s:line 17
Press any key to continue . . .
```

# Throwing and Catching Exceptions

- ◆ An exception is an error that occurs during program execution.
- ◆ An exception arises when an operation cannot be completed normally. In such situations, the system throws an error.

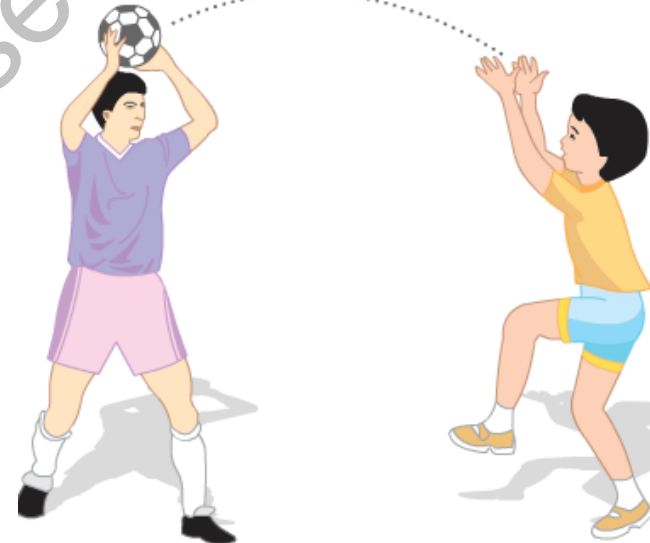


- ◆ The error is handled through the process of **exception handling**.



## Example

- ◆ Consider a group of boys playing throwball.
- ◆ If any of the boys fail to catch the ball when thrown, the game is terminated.
- ◆ Thus, the game goes on till the boys are successful in catching the ball on time.
- ◆ Similarly, in C#, exceptions that occur while working on a particular program need to be caught by exception handlers.
- ◆ If the program does not contain the appropriate exception handler, then the program might be terminated.



- ◆ 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.

- ◆ 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.

- ◆ 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);
        }
    }
}
```

- ◆ In the code:
  - ◆ The `Main()` method of the class **DivisionError** declares three variables.
  - ◆ The `try` block contains the code to divide **numOne** by **numTwo** and store the output in the **result** variable. As **numTwo** has a value of zero, the try block throws an exception.
  - ◆ This exception is handled by the corresponding catch block, which takes in **objDivide** as the instance of the `DivideByZeroException` class.
  - ◆ The exception is caught by this `catch` block and the appropriate message is displayed as the output.

### Output

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

- ◆ 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.

- ◆ 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);
        }
    }
}
```

- ◆ In the code:
  - ◆ A string array called `names` is initialized. In the `try` block, there is a statement trying to reference a fourth array element.
  - ◆ The array can store only three values, so this will cause an exception. The class **Students** consists of a general catch block declared with `Exception` and this catch block can handle any type of exception.

### Output

```
Error: System.IndexOutOfRangeException: Index was outside the bounds of the
array at
Project_New.Exception_Handling.Students.Main(String[] args) in
D:\Exception Handling\Students.cs:line 17
```

- ◆ 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 2-3

- ◆ 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);
        }
    }
}
```

- ◆ In the code:
  - ◆ The class **Employee** declares a static method called `ThrowException` that takes a string parameter called `name`.
  - ◆ If the value of `name` is `null`, the C# compiler throws the exception, which is caught by the instance of the `ArgumentNullException` class.
  - ◆ In the `try` block, the value of `name` is `null` and the control of the program goes back to the method `ThrowException`, where the exception is thrown for referencing a null value.
  - ◆ The `catch` block consists of the error handler, which is executed when the exception is thrown.

### 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

For Aptech Centre Use Only

## 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");
        }
    }
}
```

- ◆ In the code:
  - ◆ The Main() method of the class DivisionError declares and initializes two variables.
  - ◆ An attempt is made to divide one of the variables by zero and an exception is raised.
  - ◆ This exception is caught using the try-catch-finally construct.
  - ◆ The finally block is executed at the end even though an exception is thrown by the try block.

### Output

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

```
This finally block will always be executed
```

- ◆ Exceptions hinder the performance of a program.
- ◆ There are two design patterns that help in minimizing the hindrance caused due to exceptions. These are:
  - ◆ **Tester-Doer Pattern:**
    - A call that might throw exceptions is divided into two parts, tester and doer, using the tester-doer pattern.
    - The tester part will perform a test on the condition that might cause the doer part to throw an exception.
    - This test is then placed just before the code that throws an exception.



- ◆ The following code demonstrates the use of the tester-doer pattern:

### Snippet

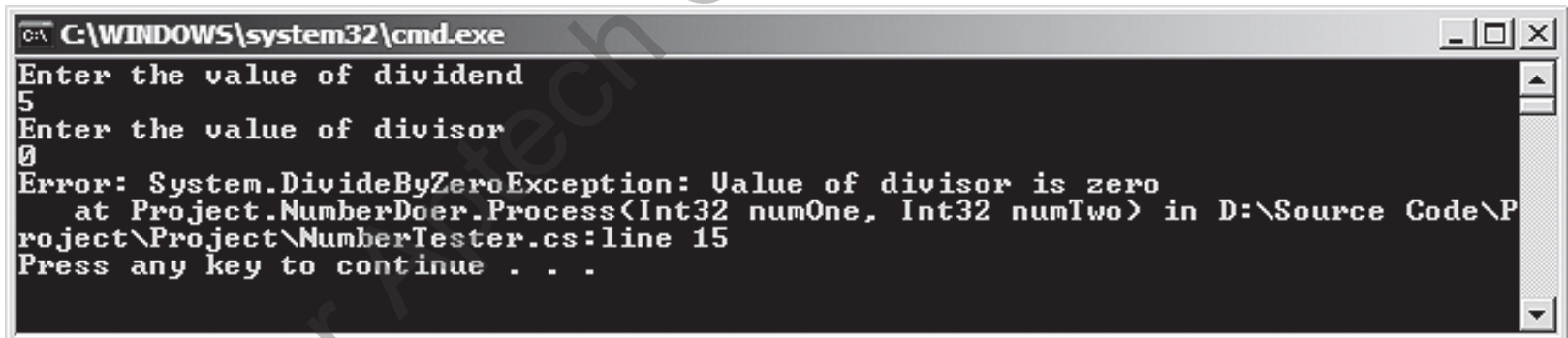
```
using System;
class NumberDoer
{
    public void Process(int numOne, int numTwo)
    {
        try
        {
            if (numTwo == 0)
            {
                throw new DivideByZeroException("Value of divisor is zero");
            }
            else
            {
                Console.WriteLine ("Quotient: " + (numOne / numTwo));
                Console.WriteLine ("Remainder: " + (numOne % numTwo));
            }
        }
        catch (DivideByZeroException objDivide)
        {
            Console.WriteLine("Error: " + objDivide);
        }
    }
}

class NumberTester
{
    NumberDoer objDoer = new NumberDoer();
    NumberDoer objDoer = new NumberDoer();
    public void AcceptDetails()
    {
    }
}
```

```
int dividend = 0;
int divisor = 0;
Console.WriteLine("Enter the value of dividend");
try {
    dividend = Convert.ToInt32(Console.ReadLine());
}
catch (FormatException objForm) {
    Console.WriteLine("Error: " + objForm);
}
Console.WriteLine("Enter the value of divisor");
try {
    divisor = Convert.ToInt32(Console.ReadLine());
}
catch (FormatException objFormat) {
    Console.WriteLine("Error: " + objFormat);
}
if ((dividend > 0) || (divisor > 0))
{
    objDoer.Process(dividend, divisor);
}
else
{
    Console.WriteLine("Invalid input");
}
}
static void Main(string[] args)
{
    NumberTester objTester = new NumberTester();
    objTester.AcceptDetails();
}
}
```



- ◆ In the code:
  - ◆ Two classes, **NumberDoer** and **NumberTester**, are defined. In the **NumberTester** class, if the value entered is not of `int` data type, an exception is thrown.
  - ◆ Next, a test is performed to check if either of the values of dividend and divisor is greater than 0. If the result is true, the values are passed to the **Process()** method of the **NumberDoer** class. In the **NumberDoer** class, another test is performed to check whether the value of the divisor is equal to 0.
  - ◆ If the condition is true, then the `DivideByZero` exception is thrown and caught by the `catch` block.
- ◆ The following figure displays the use of tester-doer pattern:



```
C:\WINDOWS\system32\cmd.exe
Enter the value of dividend
5
Enter the value of divisor
0
Error: System.DivideByZeroException: Value of divisor is zero
   at Project.NumberDoer.Process(Int32 numOne, Int32 numTwo) in D:\Source Code\Project\Project\NumberTester.cs:line 15
Press any key to continue . . .
```

## ◆ TryParse Pattern

- There are two different methods to implement the TryParse pattern. The first method is X, which performs the operation and throws the exception.
- The second method is TryX, which also performs the operation but does not throw the exception.
- Instead, it returns a boolean value that indicates whether the operation failed or succeeded.
- The following code demonstrates the use of the TryParse pattern:

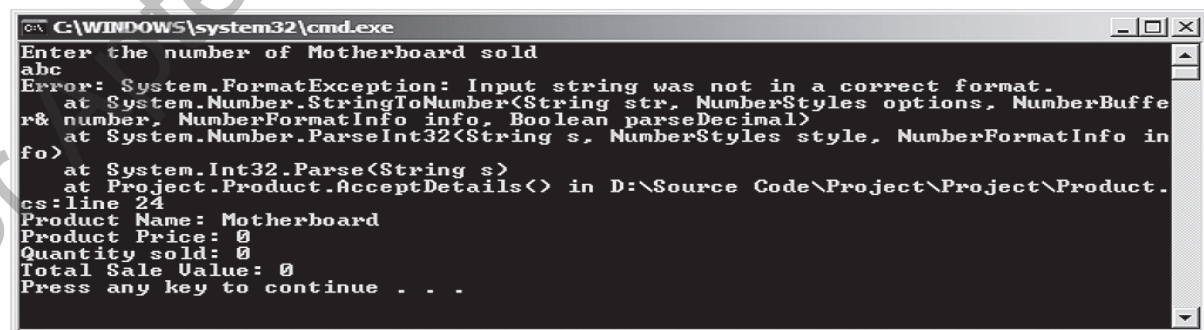
### Snippet

```
using System;
class Product
{
    private int _quantity;
    private float _price;
    private double _sales;
    string _productName;
    public Product()
    {
        _productName = "Motherboard";
    }
    public void AcceptDetails()
    {
        Console.WriteLine("Enter the number of " + _productName +
            " sold");
        try
```

```
{
    _quantity = Int32.Parse(Console.ReadLine());
}
catch (FormatException objFormat)
{
    Console.WriteLine("Error: " + objFormat);
    return;
}
Console.WriteLine("Enter the price of the product");
if (float.TryParse((Console.ReadLine()), out _price) ==
true)
{
    _sales = _price * _quantity;
}

else
{
    Console.WriteLine("Invalid price inserted");
}
}
public void Display()
{
    Console.WriteLine("Product Name: " + _productName);
    Console.WriteLine("Product Price: " + _price);
    Console.WriteLine("Quantity sold: " + _quantity);
    Console.WriteLine("Total Sale Value: " + _sales);
}
static void Main(string[] args)
{
    Product objGoods = new Product();
    objGoods.AcceptDetails();
    objGoods.Display();
}
}
```

- ◆ In the code:
  - ◆ A class **Product** is defined in which the total product sales is calculated.
  - ◆ The user-defined function **AcceptDetails** accepts the number of products sold and the price of each product. The number of products sold is converted to `int` data type using the `Parse()` method of the `Int32` structure.
  - ◆ This method throws an exception if the value entered is not of the `int` data type. Hence, this method is included in the `try...catch` block.
  - ◆ When accepting the price of the product, the `TryParse` pattern is used to verify whether the value entered is of the correct data type.
  - ◆ If the value returned by the `TryParse()` method is true, the sale price is calculated.
- ◆ The following figure displays the output of the code when the user enters a value which is not of `int` data type:



```

C:\WINDOWS\system32\cmd.exe
Enter the number of Motherboard sold
abc
Error: System.FormatException: Input string was not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   at System.Int32.Parse(String s)
   at Project.Product.AcceptDetails() in D:\Source Code\Project\Project\Product.cs:line 24
Product Name: Motherboard
Product Price: 0
Quantity sold: 0
Total Sale Value: 0
Press any key to continue . . .
  
```

## 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.



- ◆ 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.

- ◆ 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
{
    // outer finally block
}
```



## Nested try Blocks 3-4

- ◆ The following code demonstrates the use of nested try blocks:

### Snippet

```
static void Main(string[] args)
{
    string[] names = {"John", "James"};
    int numOne = 0;
    int result;
    try
    {
        Console.WriteLine("This is the outer try block");
        try
        {
            result = 133 / numOne;
        }
        catch (ArithmeticException objMaths)
        {
            Console.WriteLine("Divide by 0 " + objMaths);
        }
        names[2] = "Smith";
    }
    catch (IndexOutOfRangeException objIndex)
    {
        Console.WriteLine("Wrong number of arguments supplied " + objIndex);
    }
}
```

- ◆ In the code:
  - ◆ The array variable called **names** of type string is initialized to have two values.
  - ◆ The outer `try` block consists of another `try-catch` construct.
  - ◆ The inner `try` block divides two numbers. As an attempt is made to divide the number by zero, the inner `try` block throws an exception, which is handled by the inner `catch` block.
  - ◆ In addition, in the outer `try` block, there is a statement referencing a third array element whereas, the array can store only two values. So, the outer `try` block also throws an exception, which is handled by the outer `catch` block.

### Output

This is the outer try block

Divide by 0 System.DivideByZeroException: Attempted to divide by zero.

at Product.Main(String[] args) in  
c:\ConsoleApplication1\Program.cs:line 52

Wrong number of arguments supplied System.IndexOutOfRangeException: Index was outside the bounds of the array.

at Product.Main(String[] args) in  
c:\ConsoleApplication1\Program.cs:line 58

## Multiple catch Blocks 1-3

- ◆ 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:

### Syntax

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

- ◆ The following code demonstrates the use of multiple catch blocks:

### Snippet

```
static void Main(string[] args)
{
    string[] names = { "John", "James" };
    int numOne = 10;
    int result = 0;
    int index = 0;
    try
    {
        index = 3;
        names[index] = "Smith";
        result = 130 / numOne;
    }

    catch (DivideByZeroException objDivide)
    {
        Console.WriteLine("Divide by 0 " + objDivide);
    }
    catch (IndexOutOfRangeException objIndex)
    {
        Console.WriteLine("Wrong number of arguments supplied "
            + objIndex);
    }
    catch (Exception objException)
    {
        Console.WriteLine("Error: " + objException);
    }
    Console.WriteLine(result);
}
```

- ◆ In the code:
  - ◆ The array, **names**, is initialized to two element values and two integer variables are declared and initialized.
  - ◆ As there is a reference to a third array element, an exception of type `IndexOutOfRangeException` is thrown and the second catch block is executed.
  - ◆ Since the `try` block encounters an exception in the first statement, the next statement in the `try` block is not executed and the control terminates the `try-catch` construct.
  - ◆ So, the C# compiler prints the initialized value of the variable `result` and not the value obtained by dividing the two numbers.
  - ◆ However, if an exception occurs that cannot be caught using either of the two catch blocks, then the last catch block with the general `Exception` class will be executed.

# System.TypeInitializationException Class 1-2

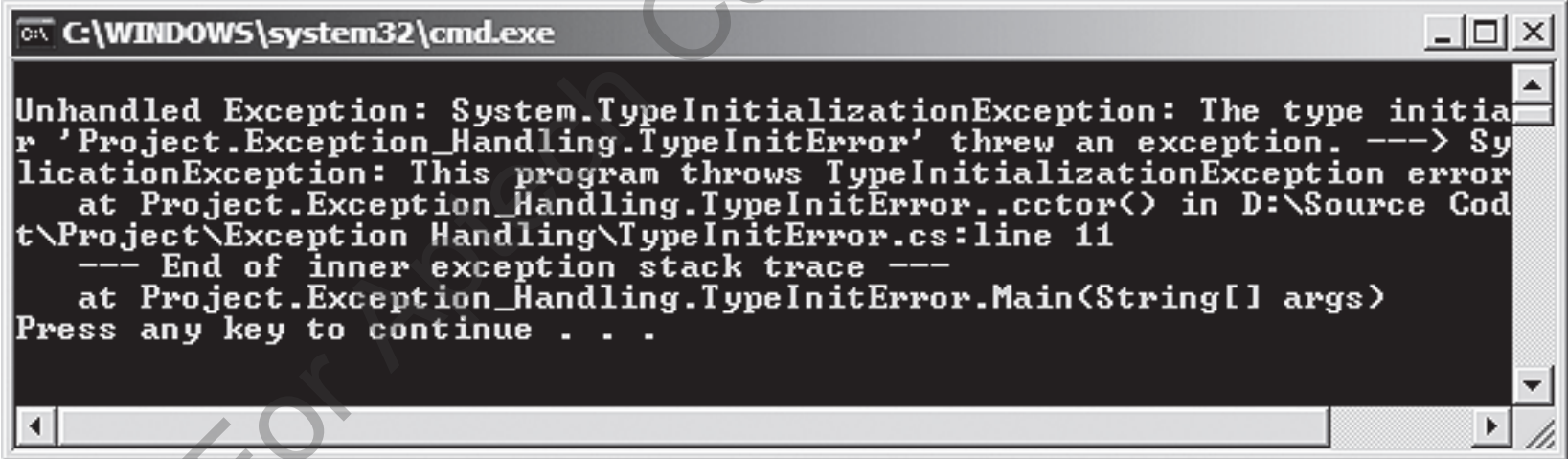
- ◆ The `System.TypeInitializationException` class is used to handle exceptions that are thrown when an instance of a class attempts to invoke the static constructor of the class as shown in the following code:

## Snippet

```
using System;
class TypeInitError
{
    static TypeInitError()
    {
        throw new ApplicationException("This program throws
        TypeInitializationException error.");
    }
    static void Main(string[] args)
    {
        try
        {
            TypeInitError objType = new TypeInitError();
        }
        catch (TypeInitializationException objEx)
        {
            Console.WriteLine("Error : {0}",objEx);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error : {0}", objEx);
        }
    }
}
```

## System.TypeInitializationException Class 2-2

- ◆ In the code:
  - ◆ A static constructor **TypeInitError** is defined. Here, only the static constructor is defined and when the instance of the class, **objType** attempts to invoke the constructor, an exception is thrown.
  - ◆ This invocation of the static constructor is not allowed and the instance of the `TypeInitializationException` class is used to display the error message in the **catch** block.
- ◆ The figure displays the `System.TypeInitializationException` generated at runtime:



```
C:\WINDOWS\system32\cmd.exe

Unhandled Exception: System.TypeInitializationException: The type initialia
r 'Project.Exception_Handling.TypeInitError' threw an exception. ---> Sy
licationException: This program throws TypeInitializationException error
  at Project.Exception_Handling.TypeInitError..cctor() in D:\Source Cod
t\Project\Exception Handling\TypeInitError.cs:line 11
  --- End of inner exception stack trace ---
  at Project.Exception_Handling.TypeInitError.Main(String[] args)
Press any key to continue . . .
```



- ◆ 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 1-2

- ◆ 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:

## Snippet

```
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);
        }
    }
}
```

### ◆ In the code:

- ◆ The class **CustomMessage** is created, which is derived from the class `Exception`.
- ◆ The constructor of the class **CustomMessage** is created and it takes a `string` parameter.
- ◆ This constructor, in turn, calls the constructor of the base class, `Exception`. The class **CustomException** consists of the `Main()` method.
- ◆ The `try` block of the class **CustomException** throws the custom exception that passes a `string` value.
- ◆ The `catch` block catches the exception thrown by the `try` block and prints the message.

### Output

This illustrates creation and catching of custom exception

- ◆ Exception assistant is a new feature for debugging C# applications. The Exception dialog box feature in earlier versions of C# is replaced by Exception assistant.
- ◆ This assistant appears whenever a run-time exception occurs.
- ◆ It shows the type of exception that occurred, the troubleshooting tips, and the corrective action to be taken, and can also be used to view the details of an exception object.
- ◆ Exception assistant provides more information about an exception than the Exception dialog box.
- ◆ The assistant makes it easier to locate the cause of the exception and to solve the problem.

- ◆ 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.