

# **Session 9:**

# **Namespaces and Exception Handling**

For Aptech Centre Use Only

# Objectives

- **Define and describe namespaces**
- **Explain nested namespaces**
- **Explain the process of throwing and catching exceptions**
- **Explain nested try and multiple catch blocks**

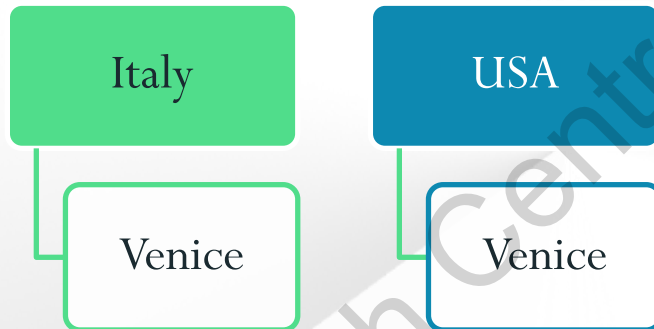
For Aptech Centre Use Only

# Introduction to Namespaces 1-3

## A namespace:

- ▶ Is an element of the C# program that helps in organizing classes, interfaces, and structures.
- ▶ Avoids name clashes between the C# classes, interfaces, and structures.

### Example



- ▶ You can easily distinguish between these two cities having same name by associating them with their respective countries.
- ▶ Similarly, in huge projects, there may be situations where classes have identical names.
- ▶ Name conflicts can be solved by having individual modules of the project use separate namespaces to store their respective classes.
- ▶ By doing this, classes can have identical names without any resultant name clashes.

# Introduction to Namespaces 2-3

```
namespace Samsung
{
    class Television
    {
        ...
    }
    class WalkMan
    {
        ...
    }
}

namespace Sony
{
    class Television
    {
        ...
    }
    class Walkman
    {
        ...
    }
}
```

Namespace

# Introduction to Namespaces 3-3

Following code renames identical classes by inserting a descriptive prefix:

## Snippet

```
class SamsungTelevision{
...
}
class SamsungWalkMan{
...
}
class SonyTelevision{
...
}
class SonyWalkMan{
...
}
```

- ▶ The identical classes **Television** and **WalkMan** are prefixed with their respective company names to avoid any conflicts.
- ▶ There can't be two classes with the same name.
- ▶ Names of the classes **become too long and become difficult to maintain.**

## Snippet

Solution by using namespaces:

```
namespace Samsung{
    class Television {
        ...
    }
    class WalkMan {
        ...
    }
}
namespace Sony{
    class Television {
        ...
    }
    class Walkman
    {
        ...
    }
}
```

- ▶ Each of the identical classes is placed in their respective namespaces, which denote respective company names.
- ▶ It can be observed that this is a **neater, better organized, and more structured way to handle naming conflicts.**



# Using Namespaces

- ▶ C# allows to specify a unique identifier for each namespace to help access classes within namespace.
- ▶ Apart from classes, following data structures can be declared in a namespace:

## Interface

- An interface is a reference type that contains declarations of the events, indexers, methods, and properties.
- Interfaces are inherited by classes and structures and all the declarations are implemented in these classes and structures.

## Structure

- A structure is a value type that can hold values of different data types.
- It can include fields, methods, constants, constructors, properties, indexers, operators, and other structures.

## Enumeration

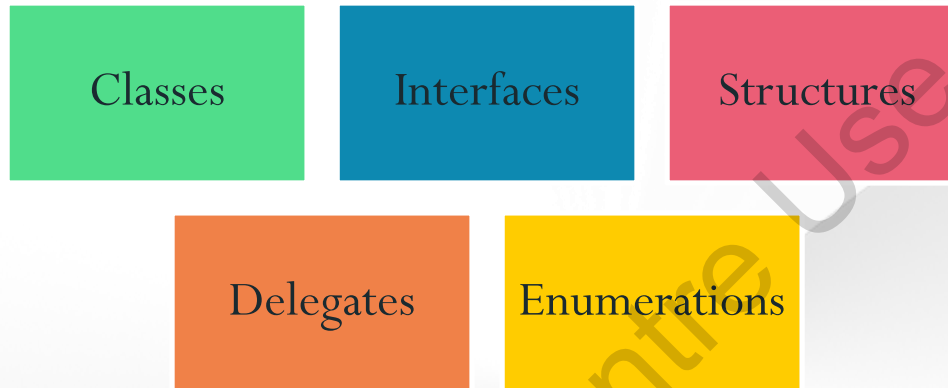
- An enumeration is a value type that consists of a list of named constants.
- This list of named constants is known as the enumerator list.

## Delegate

- A delegate is a user-defined reference type that refers to one or more methods.
- It can be used to pass data as parameters to methods.

# Built-in Namespaces 1-2

- ▶ .NET Framework comprises several **built-in** namespaces that contain:



- ▶ These namespaces are referred to as **system-defined namespaces**.
- ▶ Most commonly used built-in namespace of the .NET Framework is `System`.
- ▶ `System` namespace contains classes that:
- ▶ Define value and reference data types, interfaces, and other namespaces.
- ▶ Allow you to interact with the system, including the standard input and output devices.

# Built-in Namespaces 2-2

- Some of the most widely used namespaces within the System namespace are as follows:

## System.Collections

- Contains classes and interfaces that define complex data structures such as lists, queues, bit arrays, hash tables, and dictionaries.

## System.Data

- Contains classes that make up the ADO.NET architecture.
- The ADO.NET architecture allows you to build components that can be used to insert, modify, and delete data from multiple data sources.

## System.Diagnostics

- Contains classes that are used to interact with the system processes.
- This namespace also provides classes that are used to debug applications and trace the execution of the code.

## System.IO

- Contains classes that enable you to read from and write to data streams and files.

## System.Net

- Contains classes that allow you to create Web-based applications.

## System.Web

- Provides classes and interfaces that allow communication between the browser and the server.



# Using the System Namespace 1-4

- Two approaches of referencing the System namespace:

Class 1	Class 2
<pre>using System;</pre>	<pre>System.Console.Read(); ... ... ... System.Console.Read(); ... System.Console.Write();</pre>
<b>Efficient Programming</b>	<b>Inefficient Programming</b>

- Though both are technically valid, the first approach is more recommended.

# Using the System Namespace 2-4

- ▶ System namespace is imported by default in the .NET Framework.
- ▶ Appears in the first line of the program along with the using keyword.
- ▶ The using keyword can also be used with other namespaces whose classes you need to use.
- ▶ The using keyword eliminates the requirement to specify the name of the namespace for every class.
- ▶ Following syntax is used to access a method in a system-defined namespace:

## Syntax

```
<NamespaceName>.<ClassName>.<MethodName>;
```

Name of the namespace

Name of the class to be accessed

Name of the method within the class that is to be invoked.

- ▶ Following syntax is used to access the system-defined namespaces with the using keyword:

## Syntax

```
using <NamespaceName>;  
using <NamespaceName>.<ClassName>;
```

Name of the namespace

Name of the specific class defined in the namespace to be accessed.

# Using the System Namespace 3-4

- ▶ Following code demonstrates the use of the `using` keyword with namespaces:

## Snippet

```
using System;
class World
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World");
    }
}
```

## In the code:

- ▶ The `System` namespace is imported within the program with the `using` keyword.
- ▶ If this were not done, the program would not even compile as the `Console` class exists in the `System` namespace.

## Output

Hello World

# Using the System Namespace 4-4

- ▶ Following code refers to the Console class of the System namespace multiple times:

## Snippet

```
class World
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Hello World");
        System.Console.WriteLine("This is C# Programming");
        System.Console.WriteLine("You have executed a simple program of C#");
    }
}
```

- ▶ In the code, the class is not imported, but the System namespace members are used along with the statements. This approach is NOT recommended.

## Output

```
Hello World
This is C# Programming
You have executed a simple program of C#
```

# Custom Namespaces 1-5

- ▶ C# allows you to create namespaces with appropriate names to organize structures, classes, interfaces, and so on that can be used across different applications.
- ▶ When using a **custom** namespace, you need not worry about name clashes with classes, interfaces, and so on in other namespaces.

## Custom namespaces:

- ❑ Enable you to control the scope of a class by deciding the appropriate namespace for the class.
  - ❑ Declared using the namespace keyword and is accessed with the using keyword similar to any built-in namespace.
- ▶ Following code creates a custom namespace named Department:

### Snippet

```
namespace Department{  
    class Sales {  
        static void Main(string [] args) {  
            System.Console.WriteLine("You have created a custom namespace named  
            Department");  
        }  
    }  
}
```

## In the code:

- ❑ **Department** is declared as the custom namespace.
- ❑ The class **Sales** is declared within this namespace.

# Custom Namespaces 2-5

- ▶ Once a namespace is created, C# allows:
  - ❑ Additional classes to be included later in that namespace. Hence, namespaces are additive.
  - ❑ A namespace to be declared more than once.
- ▶ These namespaces can be split and saved in separate files or in the same file.
- ▶ At the time of compilation, these namespaces are added together.

For Aptech Centre Use Only

# Custom Namespaces 3-5

- ▶ A namespace split over multiple files is illustrated in the next three code snippets:

## Snippet

```
// The Automotive namespace contains the class SpareParts and
// this namespace is partly stored in the SpareParts.cs file.

using System;
namespace Automotive
{
    public class SpareParts
    {
        string spareName;
        public SpareParts()
        {
            _spareName = "Gear Box";
        }
        public void Display()
        {
            Console.WriteLine("Spare Part name: " + _spareName);
        }
    }
}
```

# Custom Namespaces 4-5

## Snippet

```
//The Automotive namespace contains the class Category and
// this namespace is partly stored in the Category.cs file.
using System;
namespace Automotive{
    public class Category {
        string _category;
        public Category() {
            _category = "Multi Utility Vehicle";
        }
        public void Display() {
            Console.WriteLine("Category: " + _category);
        }
    }
}
```

## Snippet

```
//The Automotive namespace contains the class Toyota and this
// namespace is partly stored in the Toyota.cs file.
namespace Automotive{
    class Toyota {
        static void Main(string[] args) {
            Category objCategory = new Category();
            SpareParts objSpare = new SpareParts();
            objCategory.Display();
            objSpare.Display();
        }
    }
}
```

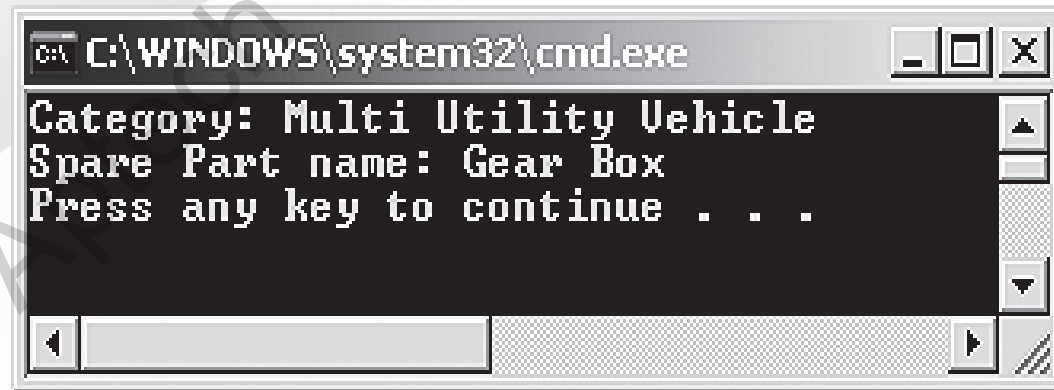


# Custom Namespaces 5-5

In the three code snippets:

- ▶ The three classes, **SpareParts**, **Category**, and **Toyota** are stored in three different files, **SpareParts.cs**, **Category.cs**, and **Toyota.cs** respectively.
  - ▶ Even though the classes are stored in different files, they are still in the same namespace, namely **Automotive**. Hence, a reference is not required here.
  - ▶ A single namespace **Automotive** is used to enclose three different classes.
  - ▶ The code for each class is saved as a separate file.
  - ▶ When the three files are compiled, the resultant namespace is still **Automotive**.
- 
- ▶ Following figure shows output of the application containing the three files:

Output



```
C:\WINDOWS\system32\cmd.exe
Category: Multi Utility Vehicle
Spare Part name: Gear Box
Press any key to continue . . .
```

# Access Modifiers for Namespaces

Namespaces are always implicitly **public**.

You cannot apply access modifiers such as `public`, `protected`, `private`, or `internal` to namespaces.

If any of the access modifiers is specified in the namespace declaration, the compiler generates an error.

# Qualified Naming 1-2

- ▶ C# allows you to use a class outside its namespace.
- ▶ A class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- ▶ This form of specifying the class is known as **Fully Qualified naming**..
- ▶ The use of fully qualified names results in long names and repetition throughout the code.
- ▶ Instead, you can access classes outside their namespaces with the using keyword.
- ▶ This makes the names short and meaningful.

For Aptech Centre Use Only

# Qualified Naming 2-2

- ▶ Following code displays the student's name, ID, subject and marks scored using a fully qualified name:

## Snippet

```
using System;
namespace Students {
    class StudentDetails {
        string _studName = "Alexander";
        int _studId = 30;
        public StudentDetails() {
            Console.WriteLine("Student Name: " + _studName);
            Console.WriteLine("Student ID: " + _studId);
        }
    }
}
namespace Examination {
    class ScoreReport {
        public string Subject = "Science";
        public int Marks = 60;
        static void Main(string[] args) {
            Students.StudentDetails objStudents = new Students.
            StudentDetails();
            ScoreReport objReport = new ScoreReport();
            Console.WriteLine("Subject: " + objReport.Subject);
            Console.WriteLine("Marks: " + objReport.Marks);
        }
    }
}
```

In the code, the class **ScoreReport** uses the class **StudentDetails** defined in the namespace **Examination**. The class is accessed by its fully qualified name.

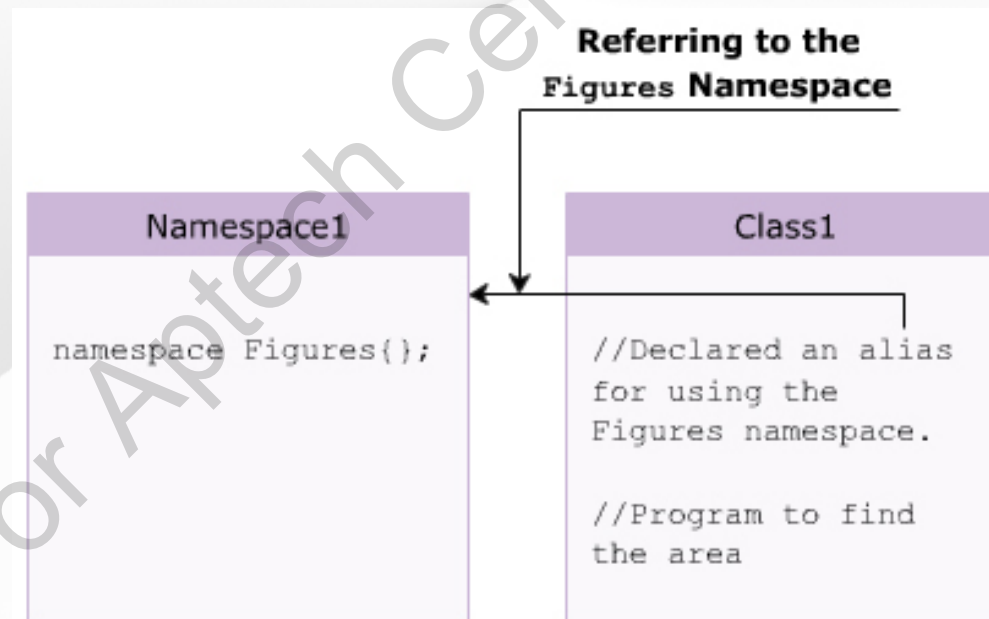
# Naming Conventions for Namespaces

- ▶ Naming conventions that should be followed for creating namespaces are:
  - ❑ Use Pascal case for naming the namespaces.
  - ❑ Use periods to separate the logical components.
  - ❑ Use plural names for namespaces wherever applicable.
  - ❑ Ensure that a namespace and a class do not have same names.
  - ❑ Ensure that the name of a namespace is not identical to the name of the assembly.



# Namespace Aliases 1-3

- ▶ Aliases are temporary alternate names that refer to the same entity.
- ▶ The namespace referred to with the `using` keyword indicates access to all the classes within the namespace.
- ▶ However, sometimes you might want to access only one class from a particular namespace.
- ▶ You can use an alias name to refer to the required class and to prevent the use of fully qualified names.
- ▶ Following figure displays an example of using namespace aliases:



# Namespace Aliases 2-3

- ▶ Following syntax is used for creating a namespace alias:

## Syntax

User-defined name assigned to the namespace

```
using <aliasName> = <NamespaceName>;
```

- ▶ Following code creates a custom namespace **Bank.Accounts.EmployeeDetails**:

## Snippet

```
namespace Bank.Accounts.EmployeeDetails {  
    public class Employees {  
        public string EmpName;  
    }  
}
```

- ▶ Following code displays the name of an employee using aliases of the `System.Console` and **Bank.Accounts.EmployeeDetails** namespaces:

## Snippet

```
using IO = System.Console;  
using Emp = Bank.Accounts.EmployeeDetails;  
class AliasExample {  
    static void Main (string[] args) {  
        Emp.Employees objEmp = new Emp.Employees();  
        objEmp.EmpName = "Peter";  
        IO.WriteLine("Employee Name: " + objEmp.EmpName);  
    }  
}
```

# Namespace Aliases 3-3

## Output

Employee Name: Peter

### In the code:

- ▶ The **Bank.Accounts.EmployeeDetails** is aliased as **Emp** and **System.Console** is aliased as **IO**.
- ▶ These alias names are used in the **AliasExample** class to access the **Employees** and **Console** classes defined in the respective namespaces.



# Namespace Alias Qualifier 1-6

- Sometimes, alias provided to a namespace may match with the name of an existing namespace. This causes an error while executing the program referencing that namespace.

```
using System;
using System.Collections.Generic;
using System.Text;
using Utility_Vehicle.Car;
using Utility_Vehicle = Automotive.Vehicle.Jeep;
```

Save as Automobile.cs  
file under the Automotive  
project

Snippet

```
namespace Automotive {
    namespace Vehicle {
        namespace Jeep {
            class Category {
                string _category;
                public Category() {
                    _category = "Multi Utility Vehicle";
                }
                public void Display() {
                    Console.WriteLine("Jeep Category: " +
                        _category);
                }
            }
        }
    }
    class Automobile {
        static void Main(string[] args) {
            Category objCat = new Category();
            objCat.Display();
            Utility_Vehicle.Category objCategory = new
            Utility_Vehicle.Category();
            objCategory.Display();
        }
    }
}
```

# Namespace Alias Qualifier 2-6

- ▶ Another project is created under the **Automotive** project to store the program shown in the following code:

## Snippet

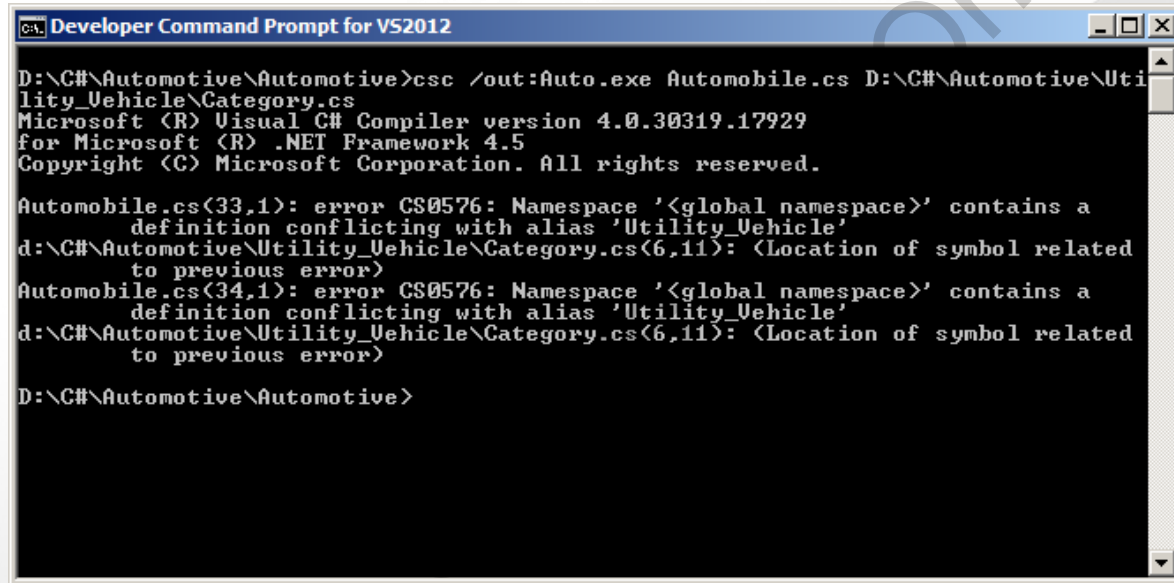
```
//The following program is saved in the Category.cs file under the
//Utility_Vehicle project created under the Automotive project.
using System;
using System.Collections.Generic;
using System.Text;

namespace Utility_Vehicle
{
    namespace Car
    {
        class Category
        {
            string _category;
            public Category()
            {
                _category = "Luxury Vehicle";
            }
            public void Display()
            {
                Console.WriteLine("Car Category: " +
                    _category);
            }
        }
    }
}
```

# Namespace Alias Qualifier 3-6

- ▶ Following figure shows the outcome of the compiled code:

## Output



```
Developer Command Prompt for VS2012
D:\C#\Automotive\Automotive>csc /out:Auto.exe Automobile.cs D:\C#\Automotive\Utility_Vehicle\Category.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

Automobile.cs(33,1): error CS0576: Namespace '<global namespace>' contains a definition conflicting with alias 'Utility_Vehicle'
d:\C#\Automotive\Utility_Vehicle\Category.cs(6,11): <Location of symbol related to previous error>
Automobile.cs(34,1): error CS0576: Namespace '<global namespace>' contains a definition conflicting with alias 'Utility_Vehicle'
d:\C#\Automotive\Utility_Vehicle\Category.cs(6,11): <Location of symbol related to previous error>

D:\C#\Automotive\Automotive>
```

## In both the codes:

- ▶ The namespaces Jeep and Car include the class Category.
- ▶ An alias Utility\_Vehicle is provided to the namespace Automotive.Jeep.
- ▶ This alias matches with the name of the other namespace in which the namespace Car is nested.
- ▶ To compile both the programs, the csc command is used which uses the complete path to refer to the Category.cs file.
- ▶ In the Automobile.cs file, the alias name Utility\_Vehicle is the same as the namespace which is being referred by the file. Due to this name conflict, the compiler generates an error.

# Namespace Alias Qualifier 4-6

- ▶ This problem of ambiguous name references can be resolved by using the namespace alias qualifier.
- ▶ Namespace alias qualifier is a new feature of C# and it can be used in the form of:

## Syntax

`<LeftOperand> :: <RightOperand>`

### In the syntax:

- ▶ `LeftOperand`: Is a namespace alias, an extern, or a global identifier.
- ▶ `RightOperand`: Is the type.

# Namespace Alias Qualifier 5-6

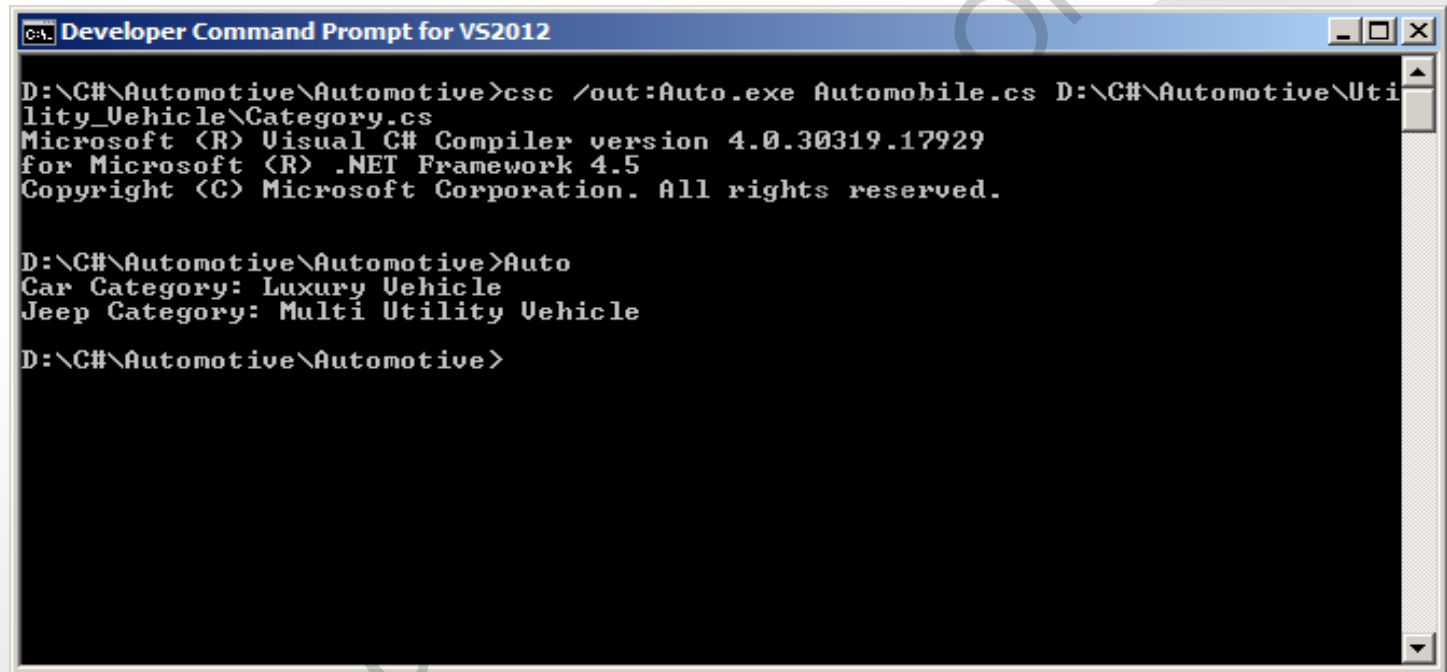
- The correct code for this is given as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using Utility_Vehicle.Car;
using Utility_Vehicle = Automotive.Vehicle.Jeep;
namespace Automotive {
    namespace Vehicle {
        namespace Jeep {
            class Category {
                string _category;
                public Category() {
                    _category = "Multi Utility Vehicle";
                }
                public void Display() {
                    Console.WriteLine("Jeep Category: " + _category);
                }
            }
        }
    }
    class Automobile {
        static void Main(string[] args) {
            Category objCat = new Category();
            objCat.Display();
            Utility_Vehicle::Category objCategory = new
            Utility_Vehicle::Category();
            objCategory.Display();
        }
    }
}
```

Snippet

# Namespace Alias Qualifier 6-6

## Output



```
C:\> Developer Command Prompt for VS2012

D:\C#\Automotive\Automotive>csc /out:Auto.exe Automobile.cs D:\C#\Automotive\Utility_Vehicle\Category.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

D:\C#\Automotive\Automotive>Auto
Car Category: Luxury Vehicle
Jeep Category: Multi Utility Vehicle

D:\C#\Automotive\Automotive>
```

### In the code:

- ▶ In the class **Automobile**, the namespace alias qualifier is used.
- ▶ The alias **Utility\_Vehicle** is specified in the left operand and the class **Category** in the right operand.

# Exceptions

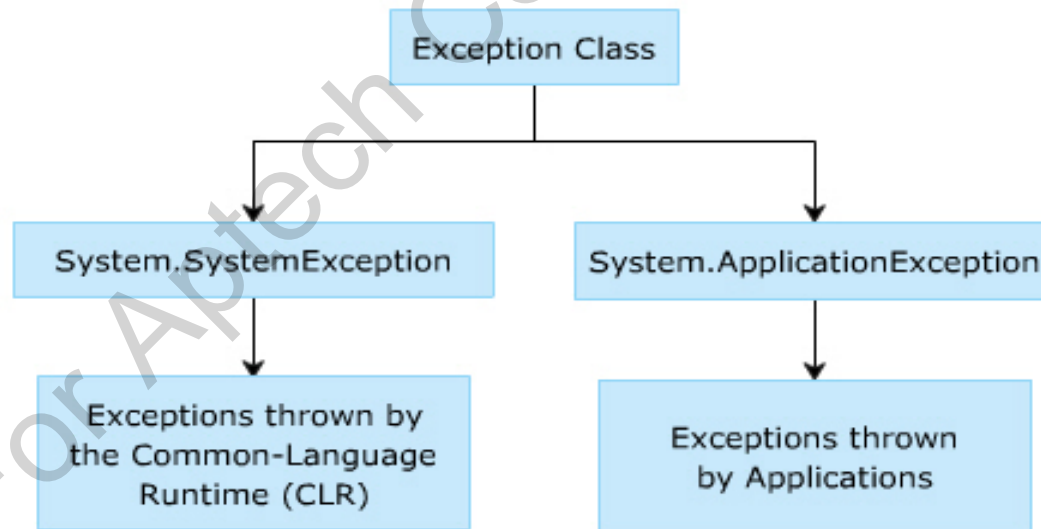
- ▶ Exceptions are abnormal events that prevent a certain task from being completed successfully.

## 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.
- ▶ Exception handling is a process of handling run-time errors.
- ▶ In C#, developers can handle exceptions by using `try-catch` or `try-catch-finally` constructs.

# Types of Exceptions

- ▶ C# allows you to handle basically two kinds of exceptions. These are as follows:
  - ❑ **System-level Exceptions:** Thrown by the system and CLR. For example, exceptions thrown due to failure in database connection or network connection are system-level exceptions.
  - ❑ **Application-level Exceptions:** Thrown by user-created applications. For example, exceptions thrown due to arithmetic operations or referencing any null object are application-level exceptions.
- ▶ Following figure displays types of exceptions in C#:

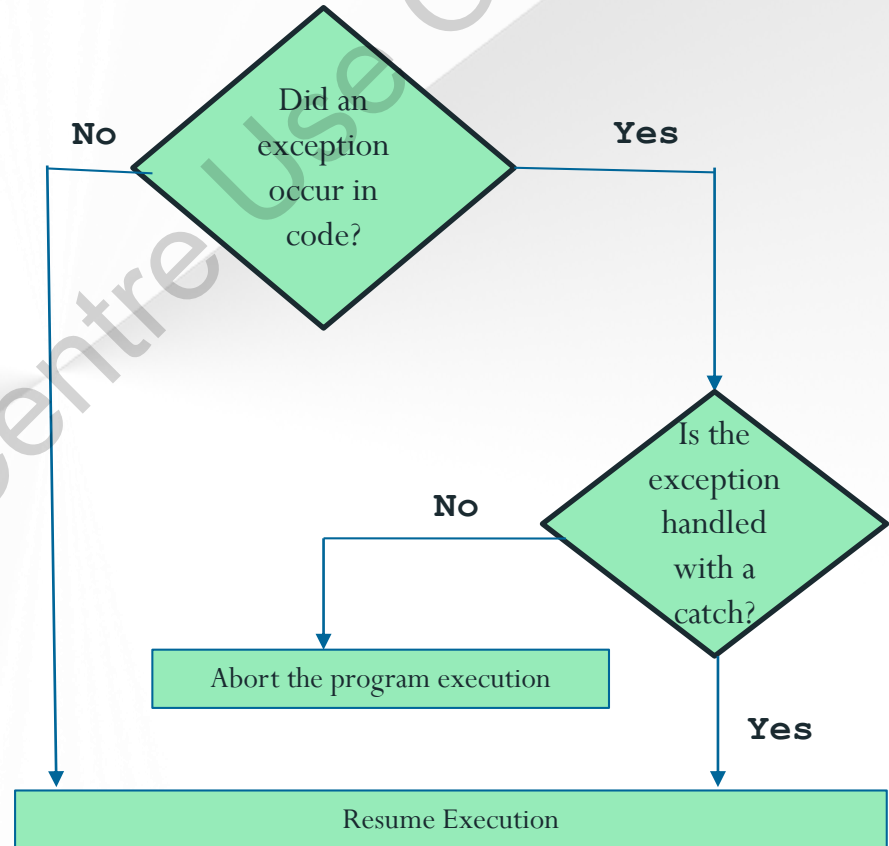




# Throwing and Catching Exceptions

## 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.



# Catching Exceptions 1-3

- ▶ Exception handling is implemented using the `try-catch` construct in C# that consists of:

`try` block

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

`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

- ▶ Following syntax is used for handling errors using `try` and `catch` blocks:

## Syntax

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

## 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);  
        }  
    }  
}
```

# Catching Exceptions 3-3

## 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)`

# 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

- ▶ 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
```

# Properties and Methods of Exception Class 1-6

- ▶ `System.Exception` is the base class that allows handling all exceptions in C#.
- ▶ All exceptions in C# inherit `System.Exception` either directly or indirectly.
- ▶ `System.Exception` contains public and protected methods that can be inherited by other exception classes.
- ▶ `System.Exception` also contains properties common to all exceptions.
- ▶ Following table describes some of the properties of `System.Exception` class:

Property	Description
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.

# Properties and Methods of Exception Class 2-6

- ▶ Following 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);
        }
    }
}
```

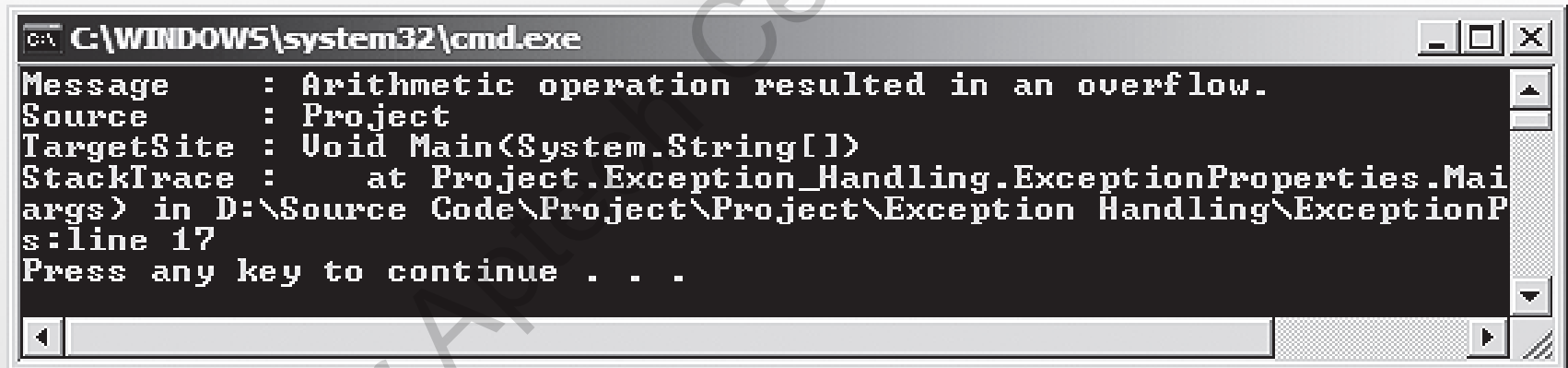


# Properties and Methods of Exception Class 3-6

## 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.

Output of the code is as follows:



```
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 . . .
```

# Properties and Methods of Exception Class 4-6

- ▶ Following table lists some 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

# Properties and Methods of Exception Class 5-6

- ▶ Following code demonstrates use of some public methods of the `System.Exception` class:

## Snippet

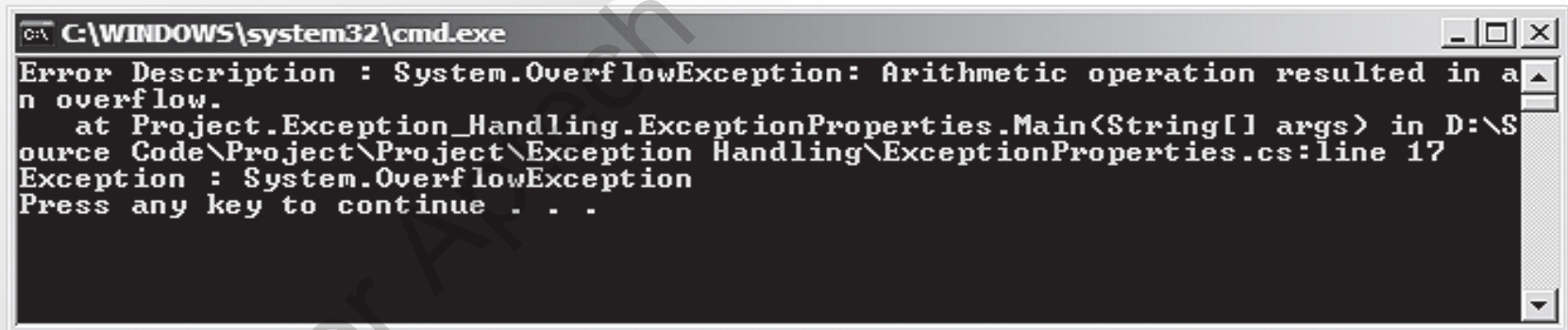
```
using System;
class ExceptionMethods {
    static void Main(string[] args) {
        byte numOne = 200;
        byte numTwo = 5;
        byte result = 0;
        try {
            result = checked((byte) (numOne * numTwo));
            Console.WriteLine("Result = {0}",
                result);
 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

# Properties and Methods of Exception Class 6-6

## 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.

Following figure displays some public methods of the `System.Exception` class:



```
C:\WINDOWS\system32\cmd.exe
Error Description : System.OverflowException: Arithmetic operation resulted in an overflow.
   at Project.Exception_Handling.ExceptionProperties.Main(String[] args) in D:\Source Code\Project\Project\Exception Handling\ExceptionProperties.cs:line 17
Exception : System.OverflowException
Press any key to continue . . .
```

# Commonly Used Exception Classes

- ▶ The `System.Exception` class has a number of derived exception classes to handle different types of exceptions.
- ▶ Following table lists some of the commonly used exception classes:

Exception	Description
<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.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 null values.
<code>System.NullReferenceException</code>	This exception is thrown while trying to assign a value to a null 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.

# InvalidCastException Class 1-2

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

## Snippet

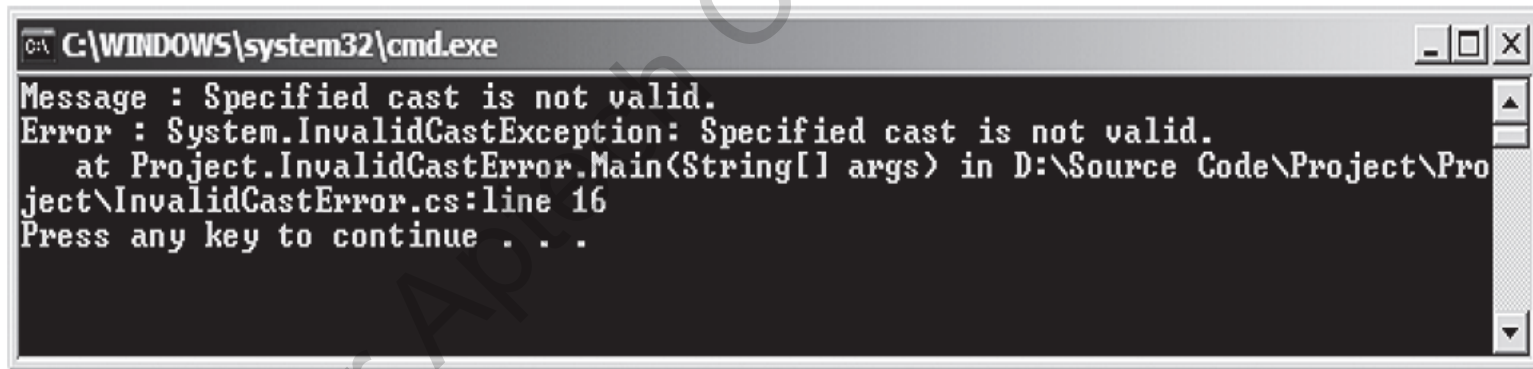
```
using System;
class InvalidCastError {
static void Main(string[] args) {
    try {
        float numOne = 3.14F;
        Object obj = numOne;
        int result = (int)obj;
        Console.WriteLine("Value of numOne = {0}", result);
    }
    catch(InvalidCastException objEx)
    {
        Console.WriteLine("Message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

# InvalidCastException Class 2-2

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#.

Following 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\Project\InvalidCastError.cs:line 16
Press any key to continue . . .
```

# NullReferenceException Class 1-2

- ▶ `NullReferenceException` is thrown when an attempt is made to operate on a null reference.
- ▶ Following code demonstrates the exception `NullReferenceException`:

## 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);
        }
    }
}
```



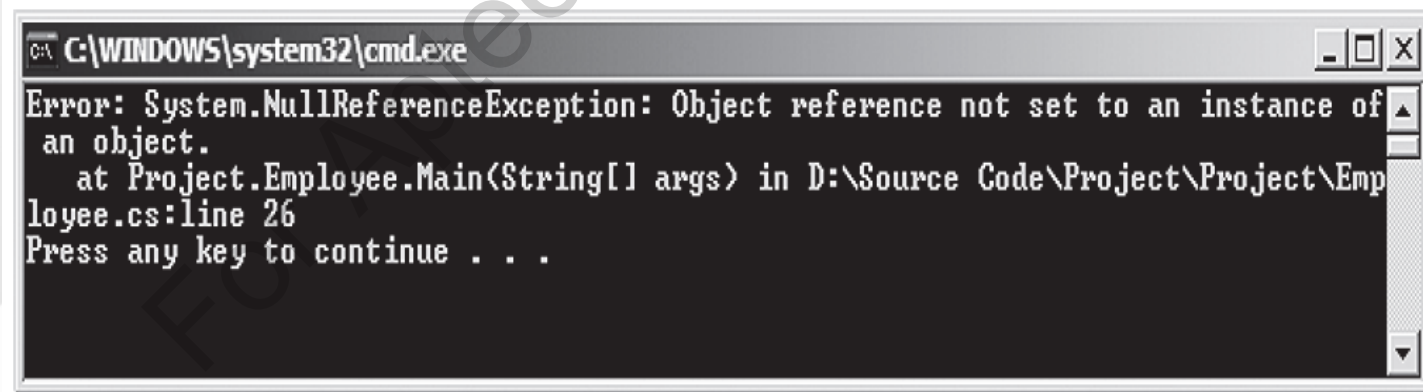
# NullReferenceException Class 2-2

```
catch (Exception objEx) {  
    Console.WriteLine("Error: " + objEx);  
}  
}  
}
```

## 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.

Following figure displays the exception that is generated when the program is executed:



# 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.
- ▶ 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);
        }
    }
}
```

# The throw Statement 2-2

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
```

# Throw Expressions

- ▶ C# 7.0 onwards has a new enhancement with respect to exceptions, namely throw expressions.
- ▶ Developers can now throw exceptions in all such code constructs.
- ▶ Following code throws an exception inside a null coalescing expression:

## Snippet

```
class Program {  
    static void Evaluate(string arg) {  
        var val = arg ?? throw new ArgumentException("Invalid  
        argument");  
        Console.WriteLine("Reached this point");  
    }  
    static void Main() {  
        Evaluate("numbers");  
        Evaluate(null);  
    }  
}
```

# 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.

For Aptech Centre Use Only

# The finally Statement 2-2

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

# 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.

For Aptech Centre Use Only

# Nested try Blocks 1-3

- ▶ 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 2-3

- ▶ 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);
    }
}
```

# Nested try Blocks 3-3

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.

For Aptech Centre Use Only

# Multiple catch Blocks 2-3

- ▶ 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);
}
```

# Multiple catch Blocks 3-3

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.

For Aptech Centres Use Only

# Exception Assistant

- ▶ 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.

# Summary

- ▶ A namespace in C# is used to group classes logically and prevent name clashes between classes with identical names.
- ▶ The `System` namespace is imported by default in the .NET Framework.
- ▶ Custom namespaces enable you to control the scope of a class by deciding the appropriate namespace for the class.
- ▶ Access modifiers such as `public`, `protected`, `private`, or `internal` are inapplicable to namespaces.
- ▶ A class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- ▶ 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.