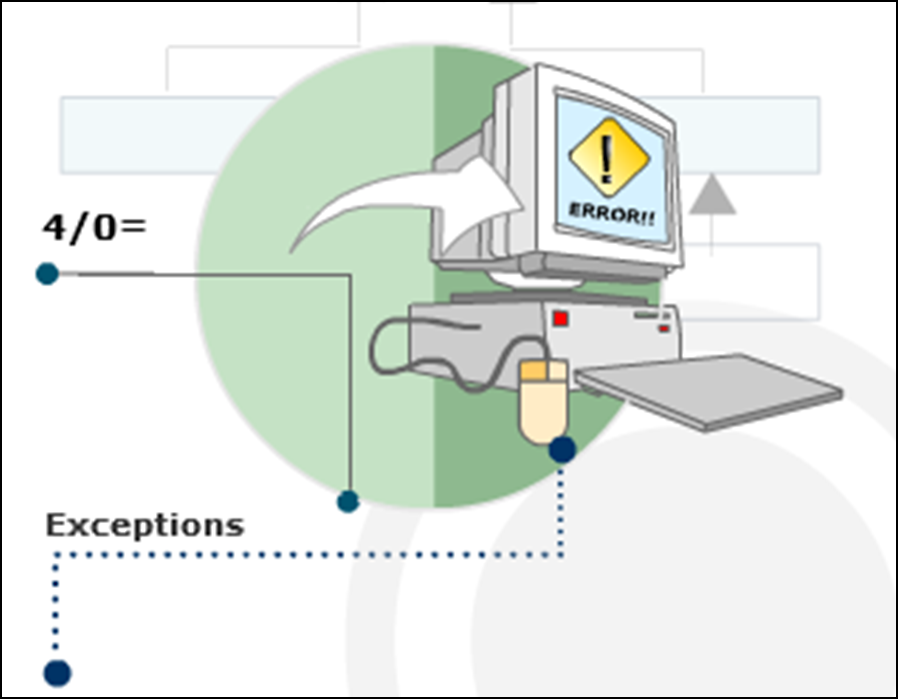
**Exception And Exception Handling In C#**

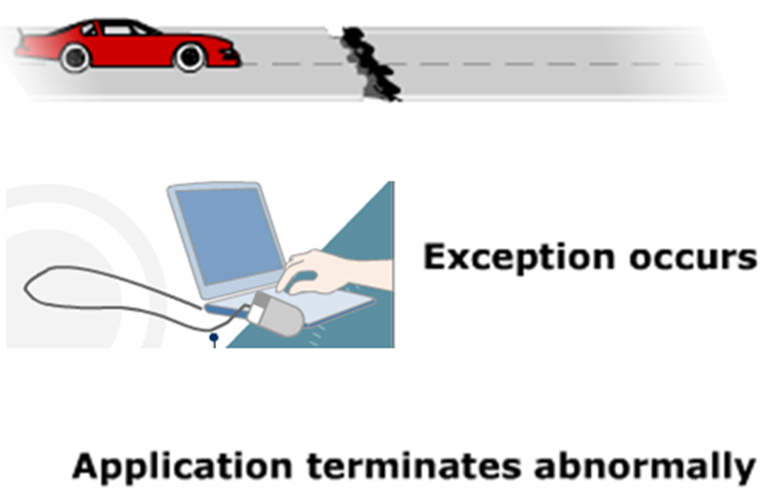
**What Is Exception?**

* An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
* In general, when a C# code encounters a situation that it cannot cope with, it raises an exception.
* An exception is a C# object that represents an error.
* When a C# code raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
* 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.

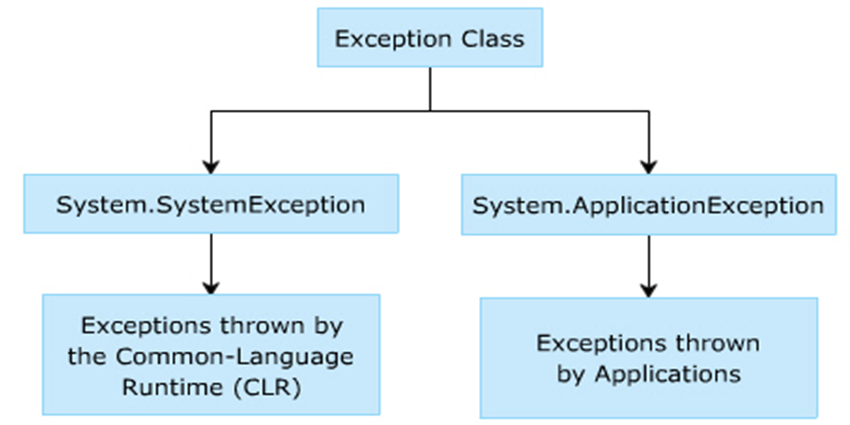


**Exceptions in C#**

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

**Types of Exceptions in C#**

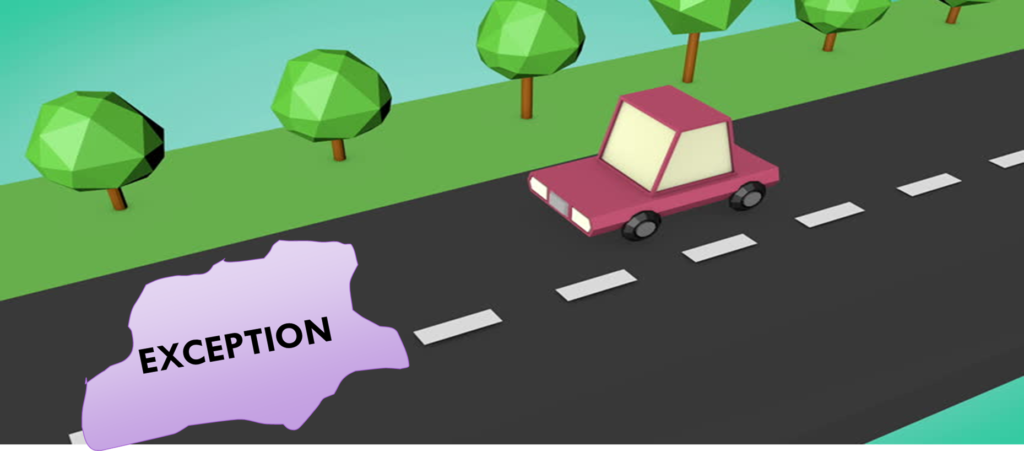
* C# can handle different types of exceptions using exception handling statements.
* It allows you to handle basically **two** kinds of exceptions. These are as follows:
  + **1. System-level Exceptions:** These are the exceptions thrown by the system that are thrown by the CLR.
  + **For example,** exceptions thrown due to failure in database connection or network connection are system-level exceptions
  + **2. 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



**When an exception occur 3 things happen**

1. Program terminates or program crashes.
2. Ugly kind of error message is displayed that user can never ever understand.
3. Statements after exception will not be executed.

**Exception Flow In Real Life**

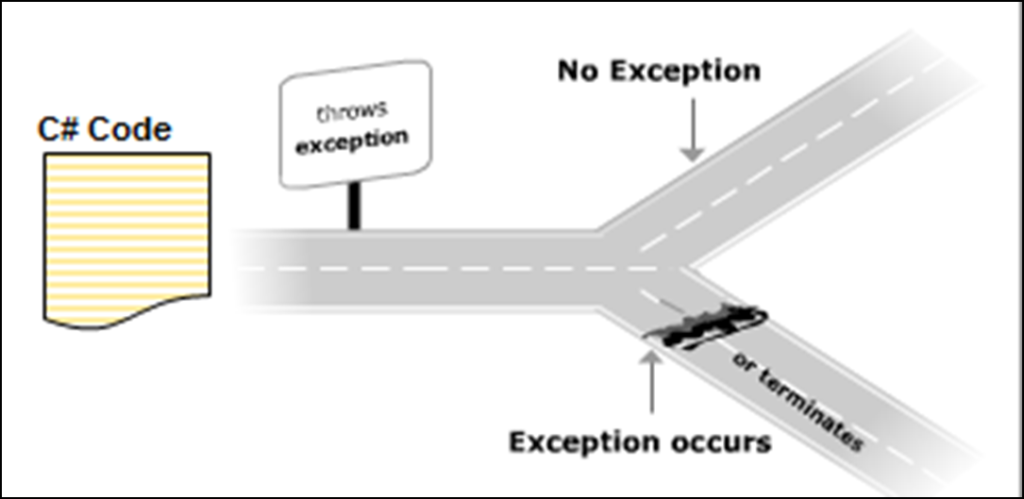


**What Is Exception?**

* An exception is a problem that arises during the execution of a program.
* A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

**What Is Exception Handling?**

* The exception handling in c# is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.



**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>)]

{

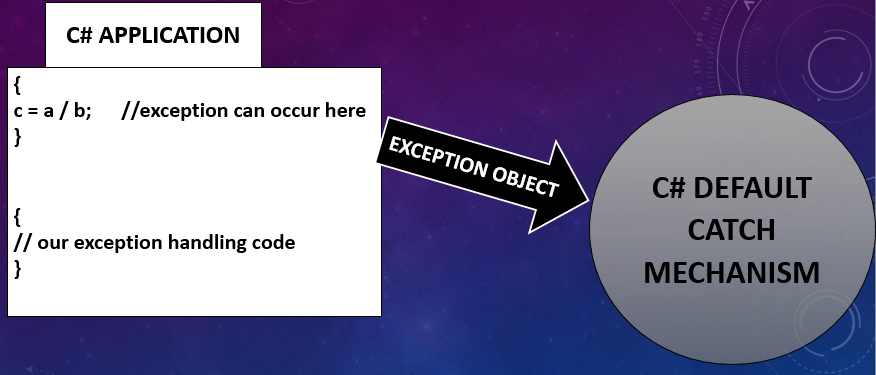
// error handling code

}

﻿

**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.



**Why We Use Exception Handling. Let's Take A Scenario:**

statement 1;  
statement 2;  
statement 3;  
statement 4;  
statement 5; // exception occurs here  
statement 6;  
statement 7;  
statement 8;  
statement 9;  
statement 10;

In above statements, if we not do exception handling then program terminates at statement no 5, and statement number 6,7,8,9,10 will not be executed.

**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:**

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 above 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

﻿

**When you do exception handling these 3 things will not occur.**

1. Program does not terminates or crashes.
2. No Ugly kind of error message is displayed, User Friendly error message is displayed.
3. Statements after exception will be executed for sure.

**About Exception Class**

**System.Exception:**

* Exception Class is the base class that allows you to handle all exceptions in C#.
* Exception Class is inherited by all exceptions in C# either directly or indirectly.
* Exception Class contains public and protected methods that can be inherited by other exception classes and also contains properties that are common to all exceptions.

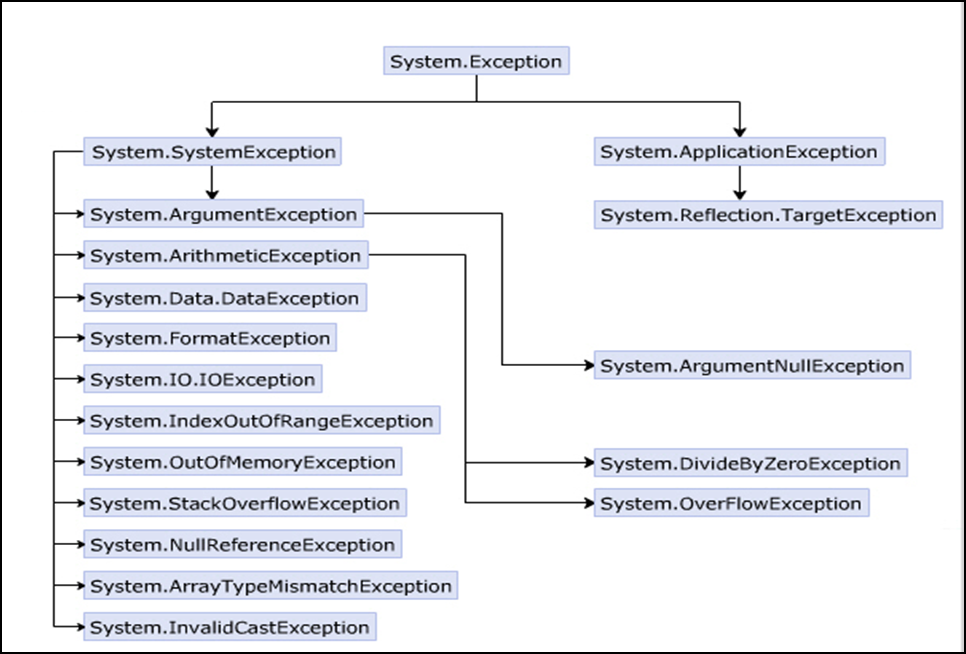
**The following table describes the properties of Exception class:**

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

﻿

**Commonly Used Exception Classes**

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



**Exception Classes**

| Exceptions | Descriptions |
| --- | --- |
| System.ArithmeticException | This exception is thrown for problems that occur due to arithmetic or casting and conversion operations. |
| System.ArgumentException | This exception is thrown when one of the arguments does not match the parameter specifications of the invoked method. |
| System.ArrayTypeMismatchException | This exception is thrown when an attempt is made to store data in an array whose type is incompatible with the type of the array. |
| System.DivideByZeroException | This exception is thrown when an attempt is made to divide a numeric value by zero. |
| System.IndexOutOfRangeException | This exception is thrown when an attempt is made to store data in an array using an index that is less than zero or outside the upper bound of the array. |
| System.InvalidCastException | This exception is thrown when an explicit conversion from the base type or interface type to another type fails. |
| System.ArgumentNullException | This exception is thrown when a null reference is passed to an argument of a method that does not accept null values. |
| System.NullReferenceException | This exception is thrown when you try to assign a value to a null object. |
| System.OutOfMemoryException | This exception is thrown when there is not enough memory to allocate to an object. |
| System.OverflowException | This exception is thrown when the result of an arithmetic, casting, or conversion operation is too large to be stored in the destination object or variable. |
| System.StackOverflowException | This exception is thrown when the stack runs out of space due to having too many pending method calls. |
| System.Data.DataException | This exception is thrown when errors are generated while using the ADO.NET components. |
| System.FormatException | This exception is thrown when the format of an argument does not match the format of the parameter data type of the invoked method. |
| System.IO.IOException | This exception is thrown when any I/O error occurs while accessing information using streams, files, and directories. |

﻿

**4 Ways Of Exception**

1. Default Throw And Default Catch
2. Default Throw And Our Catch
3. Our Throw And Default Catch
4. Our Throw And Our Catch

**The throw Statement**

* 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:

throw exceptionObject;

**where,**

* **throw:** Specifies that the exception is thrown programmatically.
* **exceptionObject:** Is an instance of a particular exception class.

**The finally Statemen**

* 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:

try  
{  
  
}  
catch()  
{  
  
}  
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 following code demonstrates the use of the try-catch-finally construct:**

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 Above 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.

**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:**

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

}

﻿

**The following code demonstrates the use of nested try blocks:**

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 Above 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.

**Try With Multiple Catch Blocks**

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

**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:**

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 Above 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.