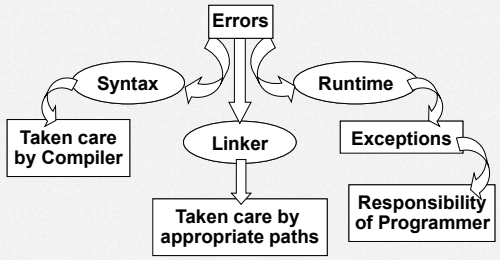
Exception Handling

C# Programming

Errors, Bugs and Exceptions

- Definitions for three commonly used anomaly-centric terms:
  - **Bugs**: These are, simply put, errors made by the programmer. For example, in C++, you fail to delete dynamically allocated memory, resulting in a memory leak, you have a bug.
  - **User errors**: User errors, on the other hand, are typically caused by the individual running your application, rather than by those who created it.
  - **Exceptions**: Exceptions are typically regarded as runtime anomalies that are difficult, if not impossible, to account for while programming your application. Possible exceptions include attempting to connect to a database that no longer exists, the programmer has little control over these "exceptional" circumstances.

Types of Error



.NET Exception Handling

- Programming with structured exception handling involves the use of four interrelated entities:
  - A class type that represents the details of the exception
  - A member that throws an instance of the exception class to the caller under the correct circumstances
  - A block of code on the caller's side that invokes the exception-prone member
  - A block of code on the caller's side that will process (or catch) the exception should it occur

What C# offers?

- The C# programming language offers four keywords
  - Try
  - Catch
  - Throw
  - Finally
- These keywords allows you to throw and handle exceptions.
- The object that represents the problem at hand is a class extending System.Exception.
- All user- and system-defined exceptions ultimately derive from the System.Exception base class

Exception Base Class Members

Properties

- Message
  - Gets a message that describes the current exception.
- Source
  - Gets or sets the name of the application or the object that causes the error.
- StackTrace
  - Gets a string representation of the frames on the call stack at the time the current exception was thrown
- TargetSite
  - Gets the method that throws the current exception.
- HelpLink
  - Gets or sets a link to the help file associated with this exception
- InnerException
  - holds a reference to the previous exception

Presented by

Ranjan Bhatnagar

Types of Errors: situations

- Compilation error, Division by constant zero

```
class Div
{
    public void Main()
    {
        int k = 10;
        k = k / 0;
        System.Console.WriteLine("hello");
    }
}
```

- DivideByZeroException at runtime

```
class Div
{
    public void Main()
    {
        int k = 10, i = 0;
        k = k / i;
        System.Console.WriteLine("hello");
    }
}
```

Defining Exception

- An exception is an abnormal condition that arises while running a program.
- Examples:
  - Attempt to divide an integer by zero causes an exception to be thrown at run time.
  - Attempt to call a method using a reference that is null.
  - Attempting to access a file that does not exist on disk.

Exception handling, required?

- If no exception handler for a given exception is available, the program stops executing with an error message. To recover from this error conditions and continue execution, error handler is required.
- To give user friendly, relevant messages when something goes wrong.
- To conduct certain critical tasks such as “save work” or “close open files/sockets” in case critical error leads to abnormal termination.
- To allow programs to terminate gracefully or operate in degraded mode.

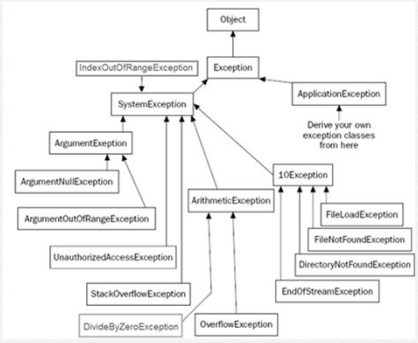
Exception handling syntax

- Handling exception that occur at runtime in our application is exception handling.
- Syntax

```
try{
        // code that may throw exception
}
catch (Exception e){
        // handler
}
finally{
        //statements
}
```

- A try block must either have a catch block or a finally block or both.

Exception Hierarchy



Try-Catch Example

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        int a, b = 0;
        try
        {
            a = 10 / b;
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine(e);
        }
    }
}
```

```
System.DivideByZeroException: Attempted to divide by zero.
   at Sample.Class1.Main(String[] args) in E:\UST\Demos\JustDemo\JustDemo\Program
.cs:line 11
Remaining program
Press any key to continue . . .
```

Presented by

Ranjan Bhatnagar

Multiple Exception

- We can write more than one catch blocks to handle different exceptions.
- Here, the two catch blocks are written to handle the **'divide by zero'** and **'index out of bounds'** exceptions.
- The DivideByZeroException and IndexOutOfRangeException classes represent these exceptions respectively.
- Since b contains a non-zero value, divide by zero exception would not get thrown. So, the first catch block would not get executed.
- The statement a [10] = 9 is exceeding the array bounds and so the second catch block would get executed.
- In the catch block we have displayed our own message because the default messages are difficult to understand. In this code two exceptions are being handled. If any one of them is raised the suitable catch block would get executed.

Example

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        int b = 2;
        int[] a = new int[5];
        try
        {
            int i = 10 / b;
            a[10] = 9;
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Divide by zero error");
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine("Index out of bounds");
        }
    }
}
```

Catch-all Exception

- Instead of writing a catch block for every possible exception we can write a catch block that accepts reference to an object of the System.Exception class.
- It is the base class of all exception classes and hence address of any type of exception object can be collected in it.
- All uncaught exception objects can be handled by a catch block which catches Exception object

Catch All

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        int b = 2;
        int[] a = new int[5];
        try
        {
            int i = 10 / b;
            a[10] = 9;
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
        }
    }
}
```

Catching non-exception objects also

- The exception filter that catch block provides must be of Exception type.
- Since .NET programs can work with several components written in other languages, what if an exception is thrown by the these components are not of Exception type.
- A catch statement without any object can be used for this purpose.
- This handler catches all the objects (Exception or non Exception objects)

Non-Exception Catch Block

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        int b = 0;
        int[] a = new int[5];
        try
        {
            int i = 10 / b;
            a[3] = 9;
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine(e);
        }
        catch
        {
            Console.WriteLine("some error occurred");
        }
    }
}
```

Presented by

Ranjan Bhatnagar

Using finally

- A try/catch scope may also define an optional finally block.
- The purpose of a finally block is to ensure that a set of code statements will always execute, exception (of any type) or not.
- For example, an exception might cause an error that terminates the current method, causing its premature return.
- However, that method may have opened a file or a network connection that needs to be closed.
- Such types of circumstances are common in programming, and C# provides a convenient way to handle them: finally.

finally

- try block can have a finally block also apart from the catch block.
- finally block will execute whether or not an exception occurs.
- It is provided so that the clean up code could be written in all cases whether an error occurs or not, like closing of a file, database connection etc.
- In C#, a try block must be followed by either a catch or finally block .

Syntax

- To specify a block of code to execute when a try/catch block is exited, include a finally block at the end of a try/catch sequence. The general form of a try/catch that includes finally is shown here:

```
try {
// block of code to monitor for errors
}
catch (ExcepType1 exOb) {
// handler for ExcepType1
}
catch (ExcepType2 exOb) {
// handler for ExcepType2
}...
finally {
// finally code
}
```

Example - finally

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        int b = 2; int[] a = new int[5];
        try{
            int i = 10 / b;
            a[10] = 9;
        }
        catch (DivideByZeroException e){
            Console.WriteLine("Divide by zero error");
        }
        catch (IndexOutOfRangeException e){
            Console.WriteLine("Index out of bounds");
        }
        finally{
            Console.WriteLine("finally");
        }
        Console.WriteLine("Remaining program");
    }
}
```

try with only finally

- It is recommended that the catch handler is provided by the method only if it knows how to handle the exception.
- Otherwise the responsibility must be left to the caller of the method.
- The try block with only finally block and no catch block is a valid syntax.
- It is useful in cases where method does not know how to handle exception but want to provide the clean up code however before the method exits

Example: try with only finally

```
using System;
class Test
{
    static void Main()
    {
        try { f(null); }
        catch (NullReferenceException n)
        {
            Console.WriteLine("Main exception :" + n.Message);
        }
    }
    static void f(string s)
    {
        try
        {
            Console.WriteLine(s.IndexOf('n'));
        }
        finally
        {
            Console.WriteLine("Bye");
        }
    }
}
```

Presented by

Ranjan Bhatnagar

Nesting try Blocks

- One try block can be nested within another.
- An exception generated within the inner try block that is not caught by a catch associated with that try is propagated to the outer try block.
- For example, here the IndexOutOfRangeException is not caught by the inner try block, but by the outer try:

```
using System;
class NestTrys
{
    static void Main()
    {
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
        try
        {
                for (int i = 0; i < numer.Length; i++)
```

Nested Try Block

```
                try {
                    Console.WriteLine(numer[i] + " / " +
                    denom[i] + " is " +
                    numer[i] / denom[i]);
                }
                catch (DivideByZeroException) {
                    Console.WriteLine("Can't divide by Zero!");
                }
            }
        }
        catch (IndexOutOfRangeException)
        {
            Console.WriteLine("No matching element found.");
            Console.WriteLine("Fatal error -- program terminated.");
        }
    }
}
```

- In the example, an exception that can be handled by the inner try—in this case a divide-by zero error—allows the program to continue.

Observation

- However, an array boundary error is caught by the outer try, which causes the program to terminate.
- Although certainly not the only reason for nested try statements, the preceding program makes an important point that can be generalized. Often, nested try blocks are used to allow different categories of errors to be handled in different ways. Some types of errors are catastrophic and cannot be fixed. Some are minor and can be handled immediately.
- Many programmers use an outer try block to catch the most severe errors, allowing inner try blocks to handle less serious ones. You can also use an outer try block as a "catch all" block for those errors that are not handled by the inner block.

Throwing an exception

- It is possible to throw an exception explicitly from code.
- This is done using throw keyword.
- Syntax:

  throw excepobject;
- Example:

```
using System;
class Test
{
    static void Main(string[] s)
    {
        double d1 = 10, d2 = s.Length;
        if (d2 == 0)
            throw new DivideByZeroException();
        d1 = d1 / d2;
    }
}
```

Example throw

```
using System;
public class Class1
{
    static void Main(string[] args)
    {
        myclass m = new myclass();
        try {
            m.fun(20);
        }
        catch (Exception e) {
            Console.WriteLine(e);
        }
    }
}
class myclass
{
    public void fun(int i)
    {
        if (i > 10)
            throw new Exception("Value out of range");
        else
            Console.WriteLine(i);
    }
}
```

Re-throw

```
using System;
class Test
{
    static void Main(string[] s)
    {
        try
        {
            int d1 = 10, d2 = s.Length;
            try { d1 = d1 / d2; }
            catch (DivideByZeroException e){
                Console.WriteLine("divide by 0");
                throw e;
            }
            catch (Exception e){
                Console.WriteLine("some exception occurred 1");
            }
            Console.WriteLine(d1);
        }
        catch (Exception e){
            Console.WriteLine("some exception occurred 2");
        }
    }
}
```

Presented by

Ranjan Bhatnagar

InnerException

- When re-throw happens, the method can choose to throw a new exception.
- In such case it is recommended that the original exception be passed to the constructor of the new **Exception** as the **InnerException** parameter.
- The constructor:

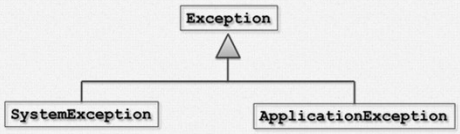
  Exception(string s, Exception innerex)
- The **property** InnerException can be used by the calling method to access the original exception that occurred.

Example - InnerException

```
using System;
class Test
{
    static void Main()
    {
        try { f(null); }
        catch (Exception n)
        {
            Console.WriteLine(n.Message);
            Console.WriteLine(n.InnerException.Message);
        }
    }
    static void f(string s)
    {
        try
        {
            Console.WriteLine(s.IndexOf('n'));
        }
        catch (NullReferenceException n)
        {
            ArgumentException ex =
            new ArgumentException("Wrong argument value: null",n);
            throw ex;
        }
        Console.WriteLine("Bye");
    }
}
```

Types of Exception

- Standard Exception
  - Exception thrown by the CLR
  - CLR throws objects of type **SystemException**
- Application Exception
  - thrown by a user program rather than the runtime.
  - Inherits from **ApplicationException**



Example Application Exception

```
using System;
class AgeException : ApplicationException
{
    string s;
    public AgeException(string str)
    {
        s = str + " is invalid age.
                        Should be between 1 and 100";
    }
    public override string ToString()
    {
        return s;
    }
}
```

Example Application Exception

```
class Test
{
    public static void Main()
    {
        try
        {
            Console.WriteLine("enter age");
            string s = Console.ReadLine();
            int num = Int32.Parse(s);
            if (num < 1 || num > 100)
                throw new AgeException(s);
        }
        catch (AgeException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Commonly encountered exceptions

- Exceptions that we have already seen
  - SystemException, NullReferenceException, DivideByZeroException, TypeInitializationException, ArgumentException, FormatException, IndexOutOfRangeException, InvalidCastException
- Other popular exceptions
  - ArithmeticException
  - OverflowException
  - OutOfMemoryException
  - StackOverflowException
  - AccessViolationException
  - ArrayTypeMismatchException

User Defined Exception

- While programming you may encounter situations when the existing predefined exceptions fall short in fulfilling our need.
- At such times we need to define our own exception classes and throw user-defined exceptions if the rules are violated.
- Our exception class should be derived from the System.Exception class.
- In the program we have declared a class called customer. In this class we have a string data member name, and two int data members, accno and balance.
- We have two methods in this class—withdraw( ) and getbalance( ). The withdraw( ) method deducts an amount, passed as a parameter, from balance and the getbalance( ) method returns the balance.

User Defined Exception

- Now we wish to apply the rule that any customer should not have a balance less than Rs. 100 in his/her account. This means an exception must be thrown whenever the customer tries to withdraw money that results in the balance to drop below Rs. 100.
- To tackle this we have written a user-defined exception class called bankexception having two int data members acc and bal.
- In the withdraw( ) method we have checked whether the balance after deduction drops below 100 or not. If it does we have raised an exception called bankexception. This exception is caught in the catch block which then proceeds to display a suitable message by calling the inform( ) method.
- If the condition in the withdraw( ) method is not satisfied then the balance is appropriately updated.

User Defined Exception Example

```
using System;
namespace Bank
{
    class customer
    {
        string name; int accno; int balance;
        public customer(string n, int a, int b)
        {
            name = n;
            accno = a;
            balance = b;
        }
        public void withdraw(int amt)
        {
            if (balance - amt <= 100)
                throw new bankexception(accno, balance);
            balance -= amt;
        }
        public int getbalance()
        {
            return balance;
        }
    }  // end of customer class
```

User Defined Exception Example

```
    class bankexception : Exception
    {
        int acc;
        int bal;
        public bankexception(int a, int b)
        {
            acc = a;
            bal = b;
        }
        public void inform()
        {
            Console.WriteLine("Account Number: " + acc
                                    + "  Balance left: " + bal);
        }
    }
```

User Defined Exception Example

```
    public class Class1
    {
        static void Main(string[] args)
        {
            customer c = new customer("Rahul", 2453, 500);
            try
            {
                c.withdraw(450);
            }
            catch (bankexception e)
            {
                Console.WriteLine("Transaction Failed ");
                e.inform();
            }
        }
    }
}
```

Building Custom Exceptions

- While you can always throw instances of System.Exception to signal a runtime error. It is sometimes advantageous to build a strongly typed exception that represents the unique details of your current problem.
- For example, assume you wish to build a custom exception named CarIsDeadException to represent the error of speeding up a doomed automobile.
- The first step is to derive a new class from System.Exception/System.ApplicationException
- As a rule, all custom exception classes should be defined as public classes. The reason is that exceptions are often passed outside of assembly
- boundaries, and should therefore be accessible to the calling code base.

Understanding Custom Exception

- Create a new Console Application project named CustomException.
- Create a Radio class and Car class as:

```
using System;
class Radio
{
    public void TurnOn(bool on)
    {
        if (on)
            Console.WriteLine("Jamming...");
        else
            Console.WriteLine("Quiet time...");
    }
}
```

Car Class

```
class Car
{
    public const int MaxSpeed = 100;
    public int CurrentSpeed { get; set; }
    public string PetName { get; set; }
    private bool carIsDead;
    private Radio theMusicBox = new Radio();
    public Car() { }
    public Car(string name, int speed)
    {
        CurrentSpeed = speed;
        PetName = name;
    }
    public void CrankTunes(bool state)
    {
        theMusicBox.TurnOn(state);
    }
```

Car Class Accelerate Method

```
    public void Accelerate(int delta)
    {
        if (carIsDead)
            Console.WriteLine("{0} is out of order...", PetName);
        else
        {
            CurrentSpeed += delta;
            if (CurrentSpeed > MaxSpeed)
            {
                CarIsDeadException ex = new CarIsDeadException(
                      string.Format("{0} has overheated!", PetName),
                                "You have a lead foot", DateTime.Now);
                ex.HelpLink = "http://www.cars.com";
                throw ex;
            }
            else
                Console.WriteLine
                      ("=> CurrentSpeed = {0}", CurrentSpeed);
        }
    }
}
```

CarIsDeadException Class

```
public class CarIsDeadException : ApplicationException
{
    private string messageDetails = String.Empty;
    public DateTime ErrorTimeStamp { get; set; }
    public string CauseOfError { get; set; }
    public CarIsDeadException() { }
    public CarIsDeadException(string message,
    string cause, DateTime time)
    {
        messageDetails = message;
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
    public override string Message
    {
        get
        {
            return string.Format
                      ("Car Error Message: {0}", messageDetails);
        }
    }
}
```

Another way

- You are not required to override the virtual Message property, as you could simply pass the incoming message to the parent's constructor as follows:

```
public class CarIsDeadException : ApplicationException
{
    public DateTime ErrorTimeStamp { get; set; }
    public string CauseOfError { get; set; }
    public CarIsDeadException() { }
    // Feed message to parent constructor.
    public CarIsDeadException
              (string message, string cause, DateTime time)
        : base(message)
    {
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
}
```

A Proper Custom Exception Class

- If you wish to build a truly prim-and-proper custom exception class, you would want to make sure your type adheres to .NET best practices. Specifically, this requires that your custom exception
  - Derives from ApplicationException
  - Is marked with the [System.Serializable] attribute
  - Defines a default constructor
  - Defines a constructor that sets the inherited Message property
  - Defines a constructor to handle "inner exceptions"
  - Defines a constructor to handle the serialization of your type

Presented by

Ranjan Bhatnagar

Proper Exception Class

```
[global::System.Serializable]
public class CarIsDeadException : ApplicationException
{
    public CarIsDeadException() { }
    public CarIsDeadException(string message) : base(message) { }
    public CarIsDeadException(string message, Exception inner)
                            : base(message, inner) { }
    protected CarIsDeadException(
      System.Runtime.Serialization.SerializationInfo info,
      System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }
    public DateTime ErrorTimeStamp { get; set; }
    public string CauseOfError { get; set; }
    public CarIsDeadException(string message,
      string cause, DateTime time) : base(message)
    {
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
}
```

Main Method

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Custom Exceptions *****\n");
        Car myCar = new Car("Rusty", 90);
        try
        {
            myCar.Accelerate(50);
        }
        catch (CarIsDeadException e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine(e.ErrorTimeStamp);
            Console.WriteLine(e.CauseOfError);
        }
        Console.ReadLine();
    }
}
```

Thanks