9 Creating And Using Exceptions

Introduction

If the reader has written C# programs that make use of the file handling facilities they will have probably written code to catch exceptions i.e. they will have used try\catch blocks. This chapter explains the importance of creating your own exceptions and shows how to do this by extending the Exception Class and using the 'Throw' mechanism.

Objectives

By the end of this chapter you will be able to….

- Appreciate the importance of exceptions
- Understand how to create your own exceptions and
- Understand how to throw these exceptions.

This chapter consists of six sections :-

1) Understanding the Importance of Exceptions
2) Kinds of Exception
3) Extending the ApplicationException Class
4) Throwing Exceptions
5) Catching Exceptions
6) Summary

9.1 Understanding the Importance of Exceptions

Exception handling is a critical part of writing C# programs. The authors of the file handling classes within the C# language knew this and created routines that made use of C# exception handling facilities – but are these really important? and do these facilities matter to programmers who write their own applications using C#?

Activity 1

Imagine part of a banking program made up of three classes, and three methods as shown below…..

The system shown above is driven by the BankManager class. The AwardLoan() method is invoked, either via the interface or from another method. This method is intended to accept or reject a loan application.

The BookofClients class maintains a set of account holders…people are added to this set if they open an account and of course they can be removed. However the only method of interest to us is the GetClient() method. This method requires a string parameter (a client ID) and either returns a client object (if the client has an account at that bank) – or returns NULL (if the client does not exist).

The Client class has only one method of interest DetermineCreditRating(). This method is invoked to determine a clients credit rating – this is used by the BankManager class to decide if a loan should be approved or not.

Considering the scenario above look at the snippet of code below …

```
Client c = listOfClients.GetClient(clientID) ;
c.DetermineCreditRating();
```

This fragment of code would exist in the AwardLoan() method. Firstly it would invoke the GetClient() method, passing a client ID as a parameter. This method would return the appropriate client object (assuming of course that a client with this ID exists) which is then stored in a local variable 'c'. Having obtained a client the DetermineCreditRating() method would be invoked on this client.

Look at these two lines of code. Can you identify any potential problems with them?

Feedback 1

If a client with the specified ID exists this code above will work. However if a client does not exist with the specified ID the GetClient() method will return NULL.

The second line of code would then cause a run time error (specifically a NullReferenceException) as it tries to invoke the DetermineCreditRating() method on a non existent client and the program would crash at this point.

Activity 2

Consider the following amendment to this code and decide if this would fix the problem.

```
Client c = listOfClients.GetClient(pClientID) ;
If (c !=NULL) {
     c.DetermineCreditRating();
}
```

Feedback 2

If the code was amended to allow for the possible NULL value returned it would work – however this protection is insecure as it relies on the programmer to spot this potential critical error.

When writing the GetClient() method the author was fully aware that a client may not be found and in this case decided to return a NULL value. However this relies on every programmer who ever uses this method to recognise and protect against this eventuality.

If any programmer using this method failed to protect against a NULL return then their program could crash – potentially in this case losing the bank large sums of money. Of course in other applications, such as an aircraft control system, a program crash could have life threatening results.

A more secure programming method is required to ensure that that a potential crash situation is always dealt with!

Such a mechanism exists - it is a mechanism called 'exceptions'.

By using this mechanism we can trap any potential errors in our code – preventing and managing crash situations.

In the situation above we could write code that would catch NullReferenceExceptions and decide how our program should respond to these. Of course we may not know specifically what generated the NullReferenceException so we may not know how our program should respond but we could at least shut down our program in a neat and tidy way, explaining to the user we are doing this because of the error generated. Doing this would be far better than allowing our program to crash without explanation.

In the example above we could be much cleverer still. The GetClient() method could be written in such a way that it generates a new type of exception 'UnkownClient' exception. We could catch this specific exception and knowing exactly what the problem is we could define a much better response…. In this case we could explain that no client exists for the specified ID, we could then ask the user to re-enter their client ID and the program could continue.

9.2 Kinds of Exception

In order to generate meaningful exceptions we need to extend the Exception base class built into the .NET framework.

The Exception class has already been extended to create a SystemException class and an ApplicationException class.

The SystemException class is used to generate exceptions within the .NET framework such as NullReferenceException.

We should use the ApplicationException class when generating exceptions within our application such as UnknownClientException.

Subclasses of ApplicationException are used to catch and deal with potential problems when running our applications. To do this we must 1) create appropriate sub classes of ApplicationException 2) generate exception objects using a **throw** clause when appropriate and 3) catch and deal with these exceptions using a **try/catch** block.

9.3 Extending the ApplicationException Class

When writing our own methods we should look for potential failure situations (e.g. value that cannot be returned, errors that may occur in calculation etc). When a potential error occurs we should generate an 'ApplicationException' object i.e. an object of the ApplicationException class. However it is best to first define a subclass of the ApplicationException i.e. to create a specialised class and throw an object of this subtype.

A new exception is just like any new class in this case it is a subclass of ApplicationException.

In the case above an error could occur if no client is found with a specified ID. Therefore we could create a new exception class called 'UnknownClientException'.

There are several overloaded constructors for the ApplicationException class. One of these requires a String and it uses this to nitialize a new instance of the ApplicationException class with a specified error message.

Exception classes have a 'Message' property we can make use of.

Thus we could create a subclass called UnknownClientException and override this constructor. When we create an object of this class we can use this string to give details of the problem that caused the exception to be generated. This string will be stored and later accessed via the Message property.

The code to create this new class is given below…..

```
public class UnknownClientException :ApplicationException
{
     public UnknownClientException(String message)
          base(message)
     {
     }
}
```

In some respects this looks rather odd. Here we are creating a subclass of ApplicationException but our subclass does not contain any new methods – nor does it override any existing methods. Thus its functionality is identical to the superclass – however it is a subtype with a meaningful and descriptive name.

If sublasses of ApplicationException did not exist we would only be able to catch the most general type of exception i.e an ApplicationException object. Thus we would only be able to write a catch block that would catch every single type of exception generated by our system.

Having defined a subclass we instead have a choice… a) we could define a catch block to catch objects of the most general type 'Exception' i.e. it would catch ALL exceptions or b) we could define a catch block that would catch ALL application generated exceptions or c) we can be more specific and catch only UnknownClientExceptions and ignore other types of exception.

By looking online at the MSDN library we can see that many predefined subclasses of SystemException already exist. There are many of these including :-

- ArithmeticException
    - DiveideByZeroException
    - OverflowException
- IOException
    - DriveNotFoundException
    - FileLoadException
    - FileNotFoundException
    - PathTooLongException
    -
- InvalidCastException
- InvalidDataException
- InvalidOperationException

Thus we could

a) write a catch block that would react to ALL of these exceptions by catching a SystemException, or we could
b) write a catch block that would catch any type of input \ output exceptions, IOExceptions, and ignore all others or
c) we could be even more specific and catch only FileNotFoundExceptions.

Catching specific exceptions allows us to take specific remedial action e.g. given a FileNotFoundException we could explain to the user of a program that the file was missing and allow them to specify an alternative file.

Catching general exceptions allows us to ensure no potentially fatal error causes our program to crash though we may not be able to take very useful remedial action e.g. given any Exception we could close our program down but without knowing what caused the exception we could not take specific remedial action.

In exactly the same way we can define sub classes of ApplicationException and catch these exception as and when appropriate.

9.4 Throwing Exceptions

Having defined our own exception classes we must then ensure our methods generate, or throw, these exceptions when appropriate. For example we would instruct a GetClient() method to throw an UnknownClientException when a client cannot be found with the specified ID.

To do this we must create an object of UnknownClientException using the keyword new. When doing so we can pass an error message as a parameter to the constructor of the exception class as shown in the code below….

```
new UnknownClientException("ClientBook.GetClient(): unknown client ID:" + clientID);
```

The code above generates an instance of the UnknownClientException class and provides the constructor with a String message. This message specifies the name of the Class / Method where the exception was generated from and provides some additional information that can inform the user about the circumstances that caused the error e.g. the clients ID.

Having generated an exception object we use the keyword 'throw' to throw this exception at the appropriate point within the body of the method.

```
public Client GetClient(int clientID)
{
     .
     .
     .
     code missing
     .
     .
     .
     throw new UnknownClientException("ClientBook.GetClient(): unknown client ID:" + clientID);

}
```

In the example above if a client is found the method will return the client object (though this code is not shown). However if a client has not been found the constructor for UnknownClientException is invoked, using 'new'. This constructor requires a String parameter – and the string we are passing here is an error message that is trying to be informative and helpful. The message is specifying :-

- the class which generated the exception (i.e. ClientBook),
- the method within this class (i.e. GetClient()),
- some text which explains what caused the exception and
- the value of the parameter for which a client could not be found.

By defining an UnkownClientException we are enabling methods calling this one to catch an deal with potentially serious errors.

9.5 Catching Exceptions

Having specified to the compiler that this method may generate an exception we are enabling other programmers to protect against potentially critical errors by placing calls to this method within a try / catch block. The code in the try block will be terminated if an exception is generated and the code in the catch block will be initiated instead.

Thus in the example above the AwardLoan() method can decide what to do if no client with the specified ID is found…..

```
try
{
      Client c = listOfClients.GetClient(clientID) ;
      c.determineCreditRating();

       // add code to award or reject a loan application based on this
          credit rating

}

catch (UnknownClientException uce)
{
     Console.WriteLine("INTERNAL ERROR IN BankManager.AwardLoan()\n"
                              + "Exception details: " + uce.Message);

}
```

Now, instead of crashing when a client with a specified ID is not found, the UnknownClientException we have deliberately thrown will be handled by the CLR engine which will terminate the code in the try block and invoke the code in the catch block, which in this case will display a message warning the user about the problem.

9.6 Summary

Exceptions provide a mechanism to deal with abnormal situations which occur during program execution.

When writing classes and methods, which may become part of a large application, we should create sub classes of the class ApplicationException and throw exception objects when appropriate.

By making use of the exception mechanism we are protecting against potentially life threatening program failure.

The exception mechanism will allow other programmers who use our methods recognise and deal with error situations.

When exceptions are generated the code in a catch block will be initiated – this code could take remedial action or terminate the program generating an appropriate error message. In either case at least the program doesn't just 'stop'.

An example of use of exceptions in a fully working system can be found in the case study Chapter 11.