Chapter 16

Exceptions

Exceptions

Objectives

After completing this unit you will be able to:

- **Use the C# exception mechanism.**
- **Create and use your own exception classes.**

Introduction to Exceptions

- **An inevitable part of programming is dealing with error conditions of various sorts.**
- **This chapter introduces the exception handling mechanism of C#, beginning with a discussion of the fundamentals of error processing and various alternatives that are available.**
- **The .NET class library provides an *Exception* class, which you can use to pass information about an exception that occurred.**
- **To further specify your exception and to pass additional information, you can derive your own class from *Exception*.**
- **When handling an exception you may want to throw a new exception.**
- **In such a case you can use the "inner exception" feature of the *Exception* class to pass the original exception on with your new exception.**
- **We will illustrate these features with a simplified version of our case study example, and then provide an update to the case study itself, incorporating basic exception handling.**
- **We also provide an example of handling arithmetic exceptions.**

Exception Fundamentals

- **The traditional way to deal with errors when programming is to have the functions you call return a status code.**
  - The status code may have a particular value for a good return and other values to denote various error conditions.
  - The calling function checks this status code, and if an error was encountered, it performs appropriate error handling.
  - This function in return may pass an error code to its calling function, and so on up the call stack.
- **Although straightforward, this mechanism has a number of drawbacks, principally lack of robustness.**
  - The called function may have impeccable error checking code and return appropriate error information, but all this information is wasted if the calling function does not make use of it.
  - The program may continue operation as if nothing were amiss, and sometime later, crash for some mysterious reason.
  - Another disadvantage is that every function in the call stack must participate in the process, or the chain of error information will be broken.
  - In languages such as C# that have constructors and overloaded operators, there is not even a return value for some operations.

.NET Exception Handling

- **C# provides an *exception* mechanism that can be used for reporting and handling errors.**
  - An error is reported by "throwing" an exception.
  - The error is handled by "catching" the exception.
  - This mechanism is similar in concept to exceptions in C++ and Java.
  - Exceptions are implemented in .NET by the Common Language Runtime, so exceptions can be thrown in one .NET language and caught in another.
- **The exception mechanism involves the following elements:**
  - Code that might encounter an exception should be enclosed in a **try** block.
  - Exceptions are caught in a **catch** block.
  - An Exception object is passed as a parameter to catch. The data type is **System.Exception** or a derived type.
  - You may have multiple **catch** blocks. A match is made based on the data type of the **Exception** object.
  - An optional **finally** clause contains code that will be executed whether or not an exception is encountered.
  - In the called method, an exception is raised through a **throw** statement.

Exception Flow of Control

- **The general structure of code which might encounter an exception is shown below:**

```
try
{
    // code that might throw an exception
}
catch (ExceptionClass1 e)
{
    // code to handle this type of exception
}
catch (ExceptionClass2 e)
{
    // code to handle this other type of exception
}
// possibly more catch handlers
// optional finally clause (discussed later)
// statements after try ... catch
```

- **Each catch handler has a parameter specifying the data type of exception that it can handle.**
- **The exception data type can be *System.Exception* or a class ultimately derived from it.**
  - If an exception is thrown, the *first* catch handler that matches the exception data type is executed, and then control passes to the statement just after the catch block(s).
  - If no handler is found, the exception is thrown to the next higher "context" (e.g., the function that called the current one). If no exception is thrown inside the **try** block, all the catch handlers are skipped.

Context and Stack Unwinding

- **As the flow of control of a program passes into nested blocks, local variables are pushed onto the stack and a new "context" is entered.**
  - Likewise, a new context is entered on a method call, which also pushes a return address onto the stack.
  - If an exception is not handled in the current context, the exception is passed to successively higher contexts until it is finally handled (or else is "uncaught" and is handled by a default system handler).
- **When the higher context is entered, C# adjusts the stack properly, a process known as *stack unwinding*.**
  - In C# exception handling, stack unwinding involves both setting the program counter and cleaning up variables (popping stack variables and marking heap variables as free, so that the garbage collector can deallocate them).

Exception Example

- **Now let's look at some code that illustrates the principles we have discussed so far.**
- **We will use a simplified version of our *Account* class, which has only methods *Deposit* and *Withdraw*, and the property *Balance*.**
  - Both methods will throw an exception if the amount passed as a parameter is negative.
  - In addition, the **Withdraw** method will throw an exception if the new balance would be negative—overdrafts are not allowed.
- **Our example program is in the directory *AccountExceptionDemo\Step1*.**
- **In the test program, we place the entire body of the command processing loop inside a *try* block.**
  - The catch handler prints an error message that is passed within the exception object.
  - Then after either normal processing or displaying an error message, a new command is read in.
  - This simple scheme provides reasonable error processing, as a bad command will not be acted upon, and the user will have an opportunity to enter a new command.

Exception Example (Cont'd)

```
// Account.cs

using System;

public class Account
{
   protected decimal balance;
   public Account(decimal balance)
   {
      this.balance = balance;
   }
   public void Deposit(decimal amount)
   {
      if (amount < 0.00m)
         throw new Exception(
"The transaction amount cannot be negative.");
      balance += amount;
   }
   public void Withdraw(decimal amount)
   {
      if (amount < 0.00m)
         throw new Exception(
"The transaction amount cannot be negative.");
      decimal newbal = balance - amount;
      if (newbal < 0.00m)
         throw new Exception(
"The balance cannot be negative.");
      balance = newbal;
   }
   public decimal Balance
   {
      get
      {
         return balance;
      }
   }
}
```

Exception Example (Cont'd)

```
// AccountExceptionDemo.cs

while (! cmd.Equals("quit"))
{
   try
   {
      if (cmd.Equals("deposit"))
      {
         decimal amount = iw.getDecimal(
            "amount: ");
         acc.Deposit(amount);
         ShowBalance(acc);
      }
      else if (cmd.Equals("withdraw"))
      {
         decimal amount = iw.getDecimal(
            "amount: ");
         acc.Withdraw(amount);
         ShowBalance(acc);
      }
      else if (cmd.Equals("show"))
         ShowBalance(acc);
      else
         help();
   }
   catch (Exception e)
   {
      Console.WriteLine(e.Message);
   }
   cmd = iw.getString("> ");
}
```

System.Exception

- **The *System.Exception* class provides a number of useful methods and properties for obtaining information about an exception.**
  - **Message** returns a text string providing information about the exception.
  - This message is set when the exception object is constructed.
  - If no message is specified, a generic message will be provided, indicating the type of the exception.
  - The **Message** property is read-only. (Hence, if you want to specify your own message, you must construct a new exception object, as done in the example above.)
  - **StackTrace** returns a text string providing a stack trace at the place where the exception arose.
  - **InnerException** holds a reference to another exception.
  - When you throw a new exception, it is desirable not to lose the information about the original exception.
  - The original exception can be passed as a parameter when constructing the new exception.
  - The original exception object is then available through the **InnerException** property of the new exception. (We will provide an example of using inner exceptions later in this chapter.)

User-Defined Exception Classes

- **You can do basic exception handling using only the base *Exception* class, as illustrated previously.**
  - In order to obtain more fine-grained control over exceptions, it is frequently useful to define your own exception class, derived from **Exception**.
  - You can then have a more specific catch handler that looks specifically for your exception type.
  - You can also define other members in your derived exception class, so that you can pass additional information to the catch handler.
- **We will illustrate by enhancing the *Withdraw* method of our *Account* class (*AccountExceptionDemo\Step1*).**
  - We want to distinguish between the two types of exceptions we throw. The one type is essentially bad input data (a negative value).
  - We will continue to handle this exception in the same manner as before (which is the same as bad input data that gives rise to a format exception, thrown by .NET library code).
  - We will define a new exception class **BalanceException** to cover the case when the balance would become negative.
  - In this case we want to allow the user an opportunity to correct the situation (in this case simply by making a deposit to cover the shortage).

User Exception Example

- **Our example program is *AccountExceptionDemo\Step2*.**
- **Note that we define a property *Shortage* that can be used to store the information about how short the balance is.**
  - The constructor of our exception class takes two parameters.
  - The first parameter is an error message string, and the second parameter is the amount of the shortage.
  - We pass the message string to the constructor of the base class.
  - We must also modify the code of the **Account** class to throw our new type of exception when an illegal negative balance would be created.
- **Finally we modify the code in our test program that processes the "withdraw" command.**
  - We place the call to **Withdraw** inside another **try** block, and we provide a catch handler for a **BalanceException**. In this catch handler we allow the user an opportunity to make a supplemental deposit.

User Exception Example (Cont'd)

```
// BalanceException.cs

using System;

public class BalanceException : Exception
{
   private decimal shortage;
   public BalanceException(string message,
      decimal shortage) : base(message)
   {
      this.shortage = shortage;
   }
   public decimal Shortage
   {
      get
      {
         return shortage;
      }
   }
}

// Account.cs

...

public void Withdraw(decimal amount)
{
   if (amount < 0.00m)
      throw new Exception(
"The transaction amount cannot be negative.");
   decimal newbal = balance - amount;
   if (newbal < 0.00m)
      throw new BalanceException(
"The balance cannot be negative.", -newbal);
   balance = newbal;
}
```

User Exception Example (Cont'd)

```
// AccountExceptionDemo.cs

...

else if (cmd.Equals("withdraw"))
{
   decimal amount = iw.getDecimal("amount: ");
   try
   {
      acc.Withdraw(amount);
   }
   catch (BalanceException e)
   {
      Console.WriteLine("You are short {0:C}",
         e.Shortage);
      Console.WriteLine("Please make a deposit");
      decimal supplemental = iw.getDecimal(
         "amount: ");
      acc.Deposit(supplemental);
      acc.Withdraw(amount);   // try again
   }
}
```

Structured Exception Handling

- **One of the principles of structured programming is that a block of code should have a single entry point and a single exit point.**
  - The **goto** statement is bad, because it facilitates breaking this principle.
  - But there are other ways to violate the principle of a single exit point, such as multiple **return** statements from a method.
  - Multiple return statements may not be too bad, because these may be encountered during normal, anticipated flow of control.
- **Exceptions can cause a particular difficulty, since they interrupt the normal flow of control.**
- **In a common scenario you can have at least three ways of exiting a method:**
  - No exception is encountered, and any catch handlers are skipped.
  - An exception is caught, and control passes to the code after the catch handlers.
  - An exception is caught, and the catch handler itself throws another exception. Then code after the catch handler will be bypassed.
- **The first two cases aren't a problem, as in both cases control passes to the code after the catch handlers, but the third case is a source of difficulty.**

Finally Block

- **The structured exception handling mechanism in C# resolves this problem with a *finally* block.**
  - The **finally** block is optional, but if present must appear immediately after the **catch** handlers.
  - It is guaranteed that the code in the **finally** block will always execute before the method is exited, in all three cases described.
- **We illustrate use of finally in the *Withdraw* command of our *Account* example.**
- **See the directory *AccountExceptionDemo\Step3*.**
  - There are several ways to exit this block of code, and the user might become confused about her balance upon exiting.
  - We insert a **finally** block, which will display the balance.

Finally Block (Cont'd)

```
// AccountExceptionDemo.cs

else if (cmd.Equals("withdraw"))
{
   decimal amount = iw.getDecimal("amount: ");
   try
   {
      acc.Withdraw(amount);
   }
   catch (BalanceException e)
   {
      Console.WriteLine(
         "You are short {0:C}", e.Shortage);
      Console.WriteLine("Please make a deposit");
      decimal supplemental = iw.getDecimal(
         "amount: ");
      acc.Deposit(supplemental);
      acc.Withdraw(amount);   // try again
   }
   finally
   {
      ShowBalance(acc);
   }
}
```

Inner Exceptions

- **In general, it is most convenient to handle exceptions near their source, because you have the most information available about the context in which the exception occurred.**
  - A common pattern is to create a new exception object that captures more detailed information and throw this on to the calling program.
- **When you throw a new exception, you don't want to lose the information about the original exception.**
- **The original exception can be passed as a parameter when constructing the new exception.**
  - The original exception is then available through the **InnerException** property of the new exception.
  - Notice that in the code above we pass the exception object e as a parameter to the constructor of the new **Exception** object that we throw.
- **In the *ArithmeticExceptionDemo* program we review checked integer arithmetic (you may wish to refer to the last section of Chapter 5), and then we present the entire example program.**

Checked Integer Arithmetic

- **By default in C#, integer overflow does not raise an exception; instead the result is truncated.**
- **The *checked* operator will cause the integer calculation to check for overflow and throw an exception if an overflow condition arises.**
- **You can cause all integer arithmetic to be checked via the */checked* compiler command line switch.**
- **You can turn off checking by the *unchecked* operator.**
- **Unchecked arithmetic is faster but less safe.**

Example Program

- **The *ArithmeticExceptionDemo* program demonstrates a number of scenarios of arithmetic exceptions.**
  - You can experiment by commenting and uncommenting different sections of code.
  - You can also try building with the **/checked** compiler option.
  - Notice how in the main program we display the inner exception, if any. (If there is no inner exception, the **InnerException** property will be **null**.)

```
public static int Main(string[] args)
{
   int prod;
   long lprod;
   try
   {
      lprod = LongMultiply(56666L, 57777L);
      Console.WriteLine("product = {0}", lprod);
      prod = Multiply(56666, 57777);
      Console.WriteLine("product = {0}", prod);
      ...
   }
   catch (Exception e)
   {
      Console.WriteLine("Exception: {0}",
         e.Message);
      if (e.InnerException != null)
         Console.WriteLine("Inner Exception: {0}",
            e.InnerException.Message);
   }
   return 0;
}
```

Summary

- **The traditional way to deal with errors in programs is to check an error return code.**
- **This approach has a number of defects, the most important of which is that the calling program may simply ignore error returns.**
- **C# provides an exception mechanism, which includes a *try* block, *catch* handlers, and a *finally* block.**
- **You can raise exceptions by means of a *throw* statement.**
- **The .NET class library provides an *Exception* class, which you can use to pass information about an exception that occurred.**
- **To further specify your exception and to pass additional information, you can derive your own class from *Exception*.**
- **When handling an exception, you may want to throw a new exception.**
- **In such a case you can use the "inner exception" feature of the *Exception* class to pass the original exception on with your new exception.**