Chapter 15

Contax T / Kodak Tri-X

Happy Mom

Exceptions: Writing Fault-Tolerant Software

Learning Objectives

- *Define the term "exception"*
- *Trace the Exception class hierarchy*
- *List and describe the runtime exceptions*
- *List and describe the properties of the Exception class*
- *Explain the difference between an application exception vs. a runtime exception*
- *Describe the exception handling mechanism supported by the .NET Common Language Runtime (CLR)*
- *Create your own custom exception classes*
- *Translate low-level exceptions into high-level exceptions*
- *State the purpose of a try/catch block*
- *State the purpose of a finally block*
- *Describe an appropriate use of a try/finally block*
- *State the proper order in which exceptions should be caught*
- *Determine what exceptions a method throws*
- *Create and throw an exception*

Introduction

C# and the .NET Common Language Runtime use exceptions to indicate that an error condition has occurred during program execution. You have been informally exposed to exceptions since Chapter 3; the code examples presented thus far have minimized the need to explicitly handle exceptions. However, C# programs are under a constant threat of running into some kind of trouble. As program complexity grows, so does the possibility that something will go wrong during its execution.

This chapter dives deep into the explanation of exceptions to show you how to properly handle existing exceptions (*i.e.,* those already defined by the .NET API) and how to create your own custom exceptions. You will be formally introduced to the Exception class, try/catch/finally blocks, and the `throw` keyword. You will also learn the difference between *application* and *runtime* exceptions. After reading this chapter, you will be well prepared to properly handle exception conditions in your programs. You will also know how to create your own custom exception classes to better fit your abstraction requirements.

Why is this chapter placed at this point in this book? The answer is simple: you will soon reach the chapters that cover topics like network and file I/O programming. In these two areas especially, you will be required to explicitly handle many types of exception conditions. As you write increasingly complex programs, you will come to rely on the information conveyed by a thrown exception to help you debug your code. Now is the time to dive deep into the topic of exceptions!

What Is An Exception

An exception is an error condition or abnormal event that occurs during program execution. By error condition, I mean a fault in the technical execution of a particular statement within your program, not bad programming logic or poor algorithms. (Although these may ultimately result in an exception!) Some examples of error conditions include the following:

- Trying to access an element beyond the bounds of an array.
- Running out of memory during program execution.
- Network communications problems.
- Unverifiable executable code modules.

Exceptions can occur in the code you write or in external code that your program calls during execution. Such code might be located in a dynamic link library (dll).

.NET CLR Exception Handling Mechanism

When an exception occurs in a program, the .NET runtime handles the problem by creating an exception object, populating it with information related to the error condition that occurred, and then passing it on to any fault-handling code that has been designated to respond to that particular type of exception. This process is referred to as *throwing* an exception.

Any code that may result in an exception condition is placed in a *protected block*. (*i.e.,* a try block) That is, if you want to provide fault handler code, (*i.e.,* a `catch` block) then you must place the suspect code in a `try` block and then handle or *catch* the thrown exception in a `catch` block.

Unhandled Exceptions

C# does not force you to use `try/catch` blocks. Unlike Java, there are no checked exceptions. The .NET runtime will try its best to find a fault handler for an exception thrown in an unprotected block of code. It does this by passing the exception object up the method call stack to see if the calling method provided a fault handler. If none is

found, the .NET runtime generates an UnhandledException event. If there are no event handlers for this Unhandled-Exception event, the .NET runtime writes a dump of the stack trace to the console and the program exits

The Exception Information Table

The .NET runtime keeps track of an executable's exception information in a data structure referred to as an *exception information table* as Figure 15-1 illustrates.

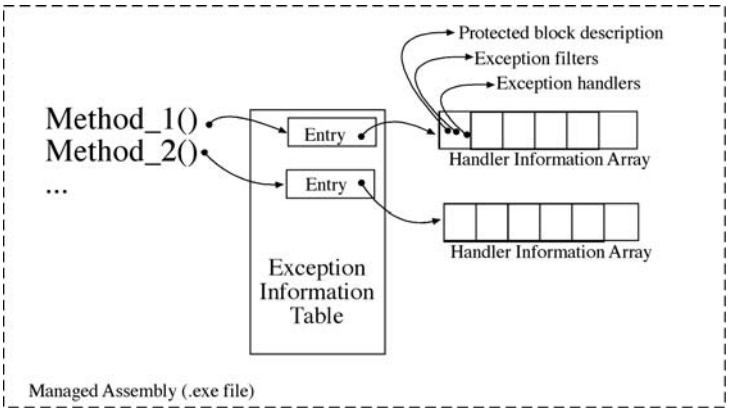

Figure 15-1: Exception Information Table

Referring to Figure 15-1 — all methods in an executable (*i.e.,* a .exe managed assembly) have an entry in this table that contains an array of exception-handling information. This array will be empty if the method does not have any `try/catch/finally` blocks. Each entry of this array contains the following information:

- A protected code block description.
- Associated exception filters.
- Exception handlers (*i.e.*, `catch` blocks, `finally` blocks, and type-filter handlers)

When an exception occurs, the .NET runtime searches the array for the first protected block of code that contains the currently executing instruction and that also has an associated exception handler. If it finds a match, the .NET runtime creates the appropriate exception-type object and then executes any `finally` or fault statements before passing the exception on to the appropriate `catch` block.

Quick Review

An *exception* is an error condition or abnormal event that occurs during program execution.

The .NET runtime handles exceptions by creating an exception object, populating it with information related to the error condition that occurred, and then passing it on to any fault-handling code that has been designated to respond to that particular type of exception. This process is referred to as *throwing* an exception.

Protected code is located in a `try` block. Error handlers are placed in a `catch` block.

Each executable file loaded into the .NET runtime contains an *exception information table*. All methods contained within the executable file have an entry in the exception information table. When an exception occurs, a method's corresponding handler information array is searched for any associated error handlers. If none are found, the exception is propagated up the method call stack to search the calling method's handler information array. If no handler methods are found, the .NET runtime dumps the stack trace to the console and the program terminates.

Exception Class Hierarchy

The .NET Framework provides a hierarchy of exception classes that derive from the System.Exception class as Figure 15-2 illustrates. Referring to Figure 15-2 — two primary types of exceptions derive from the Exception class: SystemException and ApplicationException. You'll find these exception types in the System namespace.

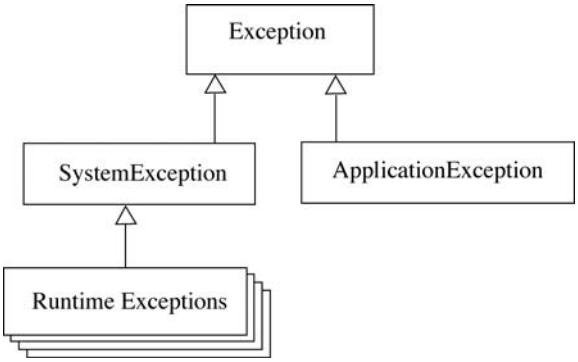

Figure 15-2: Exception Class Hierarchy

Application vs. Runtime Exceptions

Runtime exceptions derive from the SystemException class. Runtime exceptions can occur for two primary reasons: 1) failed runtime checks such as trying to access an element past the bounds of an array (IndexOutOfRangeException) or trying to access members via a reference that points to a null value (NullReferenceException), and 2) severe error conditions that occur within the runtime execution environment such as running out of memory (OutOfMemoryException) or when the execution engine has been corrupted or encounters missing data (EngineExecutionException) to name two examples. You can easily recover gracefully from an exception thrown due to a failed runtime check by catching and handling the exception, but there's not much you can do to recover from a fatal runtime exception. This is mainly because you don't know when or where in your code such an exception may occur.

An *application exception* is any exception thrown by a running program that's not a runtime exception. Application exceptions derive from either the ApplicationException class or directly from the Exception class itself. In fact, Microsoft admits that deriving custom exceptions from the ApplicationClass doesn't provide any benefit over deriving directly from Exception, so they suggest deriving all custom exceptions directly from the Exception class.

Runtime Exception Listing

Table 15-1 lists a few of the standard runtime exceptions you'll need to consider and handle.

Runtime Exception	Description
IndexOutOfRangeException	This exception is thrown if you try to access an array element that lies outside the bounds of the array. For example, assume you have an array of integers with Length 5. The array has 5 elements, 0 through Length-1. Any attempt to access an array element less than 0 or greater than 4 causes an IndexOutOfRangeException.
NullReferenceException	This exception is thrown if you try to access an object's members via a null reference. A null reference is a reference that points to nothing.
AccessViolationException	This exception is thrown if you try to access memory via an invalid pointer in unmanaged code.
InvalidOperationException	This exception is thrown if the object upon which you call a method has entered an invalid state. This usually occurs if you modify a collection and then call the Enumerator.GetNext() method on that collection. You can easily and unknowingly make this mistake by attempting to modify a collection in the body of a `foreach` loop where enumerator semantics are hidden from you.

Table 15-1: Runtime Exceptions — Partial Listing

Runtime Exception	Description
ArgumentNullException	This exception is thrown if a method does not allow one or more of its parameters to be null when the method is called.
ArgumentOutOfRangeException	This exception is thrown if one or more of a method's parameter's value falls outside of its valid value range.

Table 15-1: Runtime Exceptions — Partial Listing

Determining What Exceptions A .NET Framework Method Throws

To determine what exceptions a .NET Framework method throws, you need to consult the documentation.

Exceptions the method may throw are listed in the **Exceptions** section.

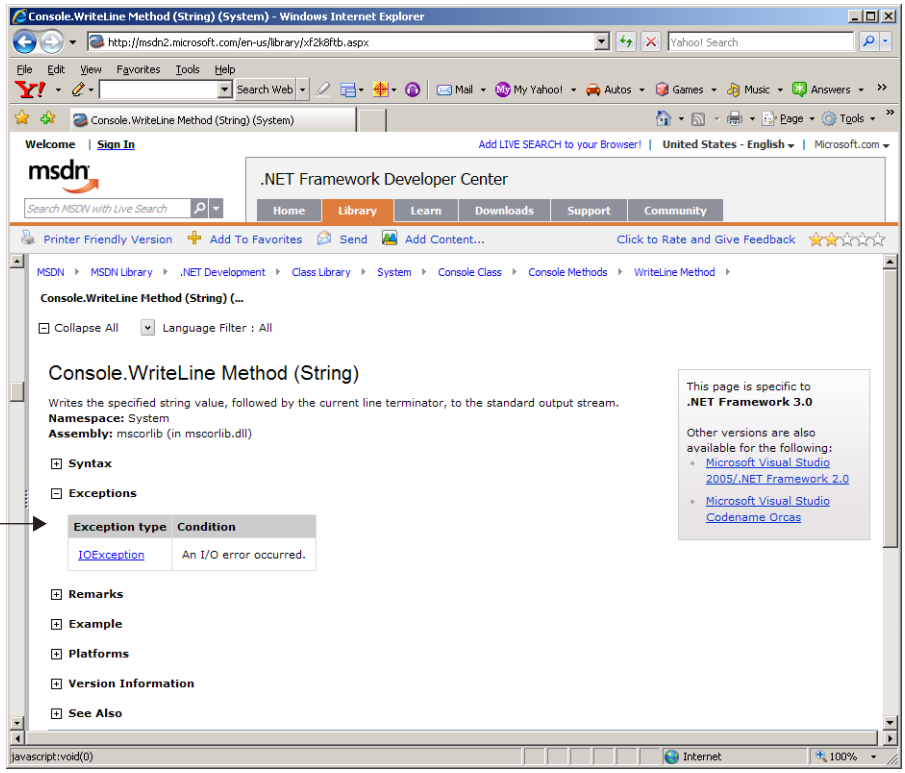

Figure 15-3: Getting Exception Information from MSDN

Referring to Figure 15-3 — you'll find a list of the exceptions a method can throw in the **Exceptions** section of the method's description page. In most cases, methods can potentially throw many different types of exceptions. It's always a good idea to check the Exception section to ensure you're properly handling any exceptions that may be thrown during a method's execution.

Quick Review

Two primary types of exceptions derive from the Exception class: *SystemException* and *ApplicationException*. Runtime exceptions can occur for two primary reasons: 1) failed runtime checks, and 2) severe error conditions that occur within the runtime execution environment. An application exception is any exception thrown by a running program that's not a runtime exception. Application exceptions derive from either the ApplicationException class or directly from the Exception class itself. Consult the **Exceptions** section of a method's MSDN documentation page to learn what exceptions the method might throw.

Exception Class Properties

A thrown exception conveys a lot of helpful information. In this section, I want to describe the properties of the Exception class. Understanding what information an exception object contains helps you to better understand what went wrong in your program. Table 15-2 lists the Exception class's public properties.

Property	Read/Write	Type	Description
Data	Readonly	Dictionary	The Data property is **readonly** and gets a collection that implements the IDictionary interface. The Data collection is by default empty. You are free to store additional information about an exception in the Data collection in the form of key/value pairs.
HelpLink	Read/Write	String	The HelpLink property is **read/write** and is used to get or set a string that represents a link to a help file or other resource that provides information about the exception.
InnerException	Readonly	Exception	The InnerException property is **readonly** and gets the exception that caused the current exception. The InnerException property is set via a constructor at the time the exception object is created.
Message	Readonly	String	Message is a **readonly** property that gets a string containing information about the current exception. The Message property is set via a constructor.
Source	Read/Write	String	The Source property is **read/write** and is used to get and set the name of the application or object that caused the exception.
StackTrace	Readonly	String	The StackTrace property is **readonly** and is used to get a string representation of the call stack at the moment the exception was generated. The StackTrace property is set automatically by the .NET runtime.
TargetSite	Readonly	MethodBase	The TargetSite property is **readonly** and is used to get information about the method that threw the exception. The TargetSite property returns a MethodBase object. The MethodBase class contains properties that describe all aspects of a particular method

Table 15-2: Exception Class Public Properties

Referring to Table 15-2 — in most cases, a property is readonly and is used to get information about a particular exception. Initializing a readonly exception property is usually done via one of the Exception class's overloaded constructors when the exception object is created. The Data property returns a dictionary collection which will always initially be empty. Populating the Data collection with relevant information is left to your discretion. The InnerException property is used when translating from one exception type to another or from low-level exceptions to higher-level exception abstractions that you create for your particular application. I cover this topic in more detail in the Creating Custom Exceptions section. The TargetSite property returns a MethodBase object, which conveys a lot of information about the method that threw the exception. You will see all of these properties in action in the next two sections.

Quick Review

The Exception class contains seven properties that are used to get information about why an exception was thrown. Most of the properties are readonly and can only be set via a constructor call when an exception object is created.

Creating Exception Handlers: Using Try/Catch/Finally Blocks

To catch and handle exceptions, you need to place exception-throwing code in a `try` block and catch the resulting exception, if and when it gets thrown, in one or more `catch` blocks where error-handling code is placed to properly deal and recover from the exception. Optionally, you can add a `finally` block whose code will always be executed regardless of whether or not an exception is thrown. This section explains the use of `try/catch/finally` blocks in detail, offers a few pointers on defensive coding, and explains in what order you must catch exceptions when using multiple `catch` blocks.

Using A Try/Catch Block

The use of a `try/catch` block is straightforward. You place any code that may throw an exception within the body of the `try` block and follow it with one or more `catch` blocks. The code that resides within the body of a `try` block is referred to as *protected code*.

A `catch` block contains error-handling code that is executed if an exception of the type the `catch` block is waiting for is thrown. Example 15.1 demonstrates the use of a `try/catch` block to protect against possible problems associated with processing console arguments.

15.1 ConsoleArgs.cs

```
1    using System;
2
3    public class ConsoleArgs {
4       public static void Main(String[] args){
5          try{
6
7             Console.WriteLine(args[0]);
8
9          }catch(IndexOutOfRangeException e){
10            Console.WriteLine("HelpLink:" + e.HelpLink);
11            Console.WriteLine("Message:" + e.Message);
12            Console.WriteLine("Source:" + e.Source);
13            Console.WriteLine("TargetSite:" + e.TargetSite.Name);
14            Console.WriteLine("StackTrace:" + e.StackTrace);
15         }
16      }
17   }
```

Referring to Example 15.1 — this program prints to the console the first argument of the args array. The problem with this program is that it may be run without any arguments and therefore the args array will be empty. Any attempt to access an element of an args array that contains no elements results in an IndexOutOfRangeException. To handle this possibility, the code on line 7 is placed within the protection of a `try` block.

The `catch` block that begins on line 9 specifically targets the IndexOutOfRangeException. A parameter of this type named *e* is declared for use within the body of the `catch` block. The code within the body of the `catch` block demonstrates the use of the various Exception properties. In practice, you don't always need to print this much information about an exception to the console.

Figure 15-4 shows the results of running this program.

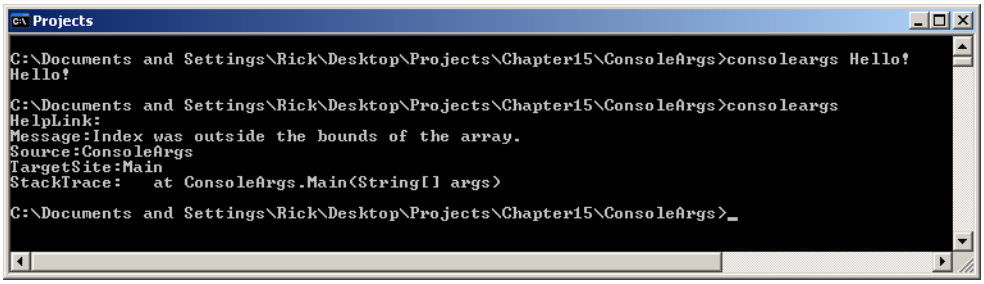

Figure 15-4: Results of Running Example 15.1

First Line of Defense: Use Defensive Coding

It's often a good idea to avoid the possibility of throwing an exception in the first place. It's good coding practice to apply defensive coding techniques to detect the possibility of an exception condition and handle the situation accordingly. Note that this is not always possible, especially for certain types of runtime exceptions.

Example 15.2 shows how a slight modification to the ConsoleArgs code can avoid the possibility of throwing an IndexOutOfRangeException altogether.

15.2 ConsoleArgs.cs (Mod 1)

```
1    using System;
2
3    public class ConsoleArgs {
4       public static void Main(String[] args){
5          try{
6
7             if(args.Length > 0){
8                Console.WriteLine(args[0]);
9             }
10
11         }catch(IndexOutOfRangeException e){
12            Console.WriteLine("HelpLink:" + e.HelpLink);
13            Console.WriteLine("Message:" + e.Message);
14            Console.WriteLine("Source:" + e.Source);
15            Console.WriteLine("TargetSite:" + e.TargetSite.Name);
16            Console.WriteLine("StackTrace:" + e.StackTrace);
17         }
18      }
19   }
```

Referring to Example 15.2 — the `if` statement beginning on line 7 checks to see if args.Length is greater than zero. If it is, it must contain at least one argument string, otherwise, it's not accessed.

Figure 15-5 shows the results of running this program.

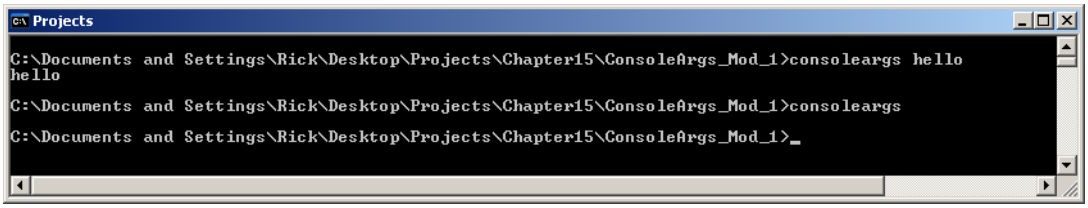

Figure 15-5: Results of Running Example 15.2

Using Multiple Catch Blocks

If your code can potentially throw several types of exceptions you can add multiple `catch` blocks, one for each exception type. The rule to follow when using multiple `catch` blocks is to catch the most specific exception(s) first, and catch the most general exception(s) last. Example 15.3 offers an example of using multiple `catch` blocks. This program converts command-line arguments to integers, adds them, and prints the sum to the console.

15.3 CommandLineAdder.cs

```
1    using System;
2
3    public class CommandLineAdder {
4       public static void Main(String[] args){
5          try{
6             int total = 0;
7             for(int i=0; i<args.Length; i++){
8                total += Int32.Parse(args[i]);
9             }
10            Console.WriteLine("You entered {0} arguments and their total comes to {1}", args.Length, total);
11         }catch(FormatException){
12            Console.WriteLine("One or more arguments failed to convert to an integer!");
13         }catch(IndexOutOfRangeException){
14            Console.WriteLine("No command line arguments entered!");
15         }catch(Exception){
16            Console.WriteLine("The program encountered an unknown problem...");
17         }
18      }
19   }
```

Referring to Example 15.3 — the `try` block in this example has three accompanying `catch` blocks which handle a range of possible exceptions including the granddaddy of them all, Exception. Note that you're not required to declare a formal parameter in the `catch` block if you're not planning on manipulating the exception object. Figure 15.6 shows the results of running this program.

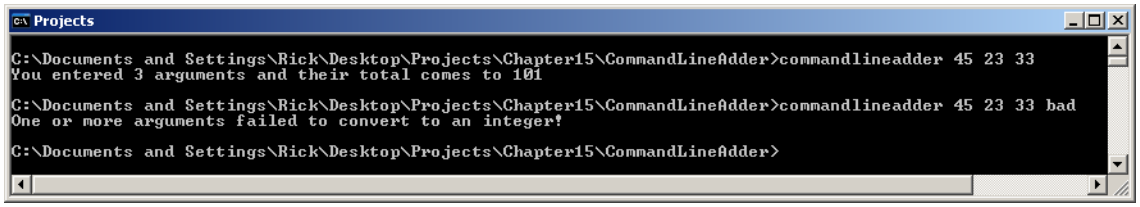

Figure 15-6: Results of Running Example 15.3

Using A Finally Block

Referring to Example 15.3 and Figure 15-6 above, note that when a bad string (*i.e.*, one that does not parse to an integer) is entered on the command line, an exception is thrown and line 10 is skipped. All code that comes after an exception-throwing statement in a `try` block is skipped. Depending on the nature of the application, this may or may not be acceptable.

In certain programming situations, you will want to ensure that a certain piece of code executes in all cases regardless of whether or not an exception is thrown. A `finally` block is used for just this purpose. Example 15.4 demonstrates the use of a finally block.

15.4 CommandLineAdder.cs (Mod 1)

```
1    using System;
2
3    public class CommandLineAdder {
4       public static void Main(String[] args){
5          try{
6             int total = 0;
7             for(int i=0; i<args.Length; i++){
8                total += Int32.Parse(args[i]);
9             }
10            Console.WriteLine("You entered {0} arguments and their total comes to {1}", args.Length, total);
11         }catch(FormatException){
12            Console.WriteLine("One or more arguments failed to convert to an integer!");
13         }catch(IndexOutOfRangeException){
14            Console.WriteLine("No command line arguments entered!");
15         }catch(Exception){
16            Console.WriteLine("The program encountered an unknown problem...");
17         }finally{
18            Console.WriteLine("Thank you for using Command Line Adder!");
19         }
20      }
21   }
```

Referring to Example 15.4 — the `finally` block starts on line 17 and simply ensures that no matter what happens when the program runs, the "Thank you..." message gets printed to the console. You will see plenty of good uses for a `finally` block when you study file input/output (I/O) and network programming. Figure 15-7 shows the results of running this program.

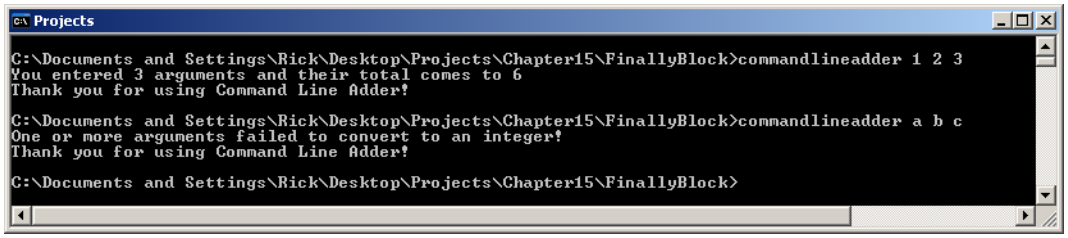

Figure 15-7: Results of Running Example 15.4

Quick Review

To catch and handle exceptions, you need to place exception-throwing code in a `try` block and catch the resulting exception, if and when it gets thrown, in one or more `catch` blocks where error-handling code is placed to properly deal and recover from the exception. Optionally, you can add a `finally` block whose code will always be executed regardless of if an exception is thrown.

All code within a `try` block that follows an exception-throwing statement is skipped. Place critical resource releasing code in a `finally` block.

If code can potentially throw several types of exception, add multiple `catch` blocks, one for each exception type. The rule to follow when using multiple `catch` blocks is to catch the most specific exception(s) first, and catch the most general exception(s) last.

Creating Custom Exceptions

Although the .NET Framework API offers lots of different types of exception classes, you'll eventually have a need to create your own custom exceptions to better model the types of things that can go wrong in your application. This section shows you how to create custom exception classes by extending the Exception class. I also show you how to manually throw an exception and how to translate low-level exceptions into higher-level exceptions.

Extending The Exception Class

As you learned previously in this chapter, although there exists an ApplicationException class, Microsoft suggests that you instead derive custom exception classes directly from the Exception class. At a minimum, you'll want to end the name of your custom exception class with the word "Exception" and provide an implementation for each of the three common Exception constructors. Example 15.5 gives the code for a custom exception named SurrealistNotFoundException, which is used later in this section to demonstrate how to manually throw an exception.

15.5 SurrealistNotFoundException.cs

```
1    using System;
2
3    public class SurrealistNotFoundException : Exception {
4
5       public SurrealistNotFoundException() : base("Surrealist not found!") { }
6
7       public SurrealistNotFoundException(String message) : base(message) { }
8
9       public SurrealistNotFoundException(String message, Exception inner_exception)
10                                      : base(message, inner_exception) { }
11
12   }
```

Referring to Example 15.5 — this custom exception provides implementations for the three basic Exception constructors. The default constructor that begins on line 3 supplies a default message. The rest of the code is fairly straightforward and easy to follow. Let's now see how this custom exception might be used in a program. Examples 15.6 and 15.7 give the code for a short program that allows users to look up the names of famous surrealists.

15.6 SurrealistBank.cs

```
1    /*************************************************
2      This program depends on the Person class.
3    *************************************************/
4
5    using System;
6    using System.Collections.Generic;
7
8    public class SurrealistBank {
9       Dictionary<String, Person> surrealists;
10
11      public SurrealistBank() {
12         surrealists = new Dictionary<String, Person>();
13         InitializeDictionary();
14   }
15
16      private void InitializeDictionary() {
17         Person p1 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
18         Person p2 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
```



```
19        Person p3 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
20        Person p4 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
21        Person p5 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
22                               new DateTime(1887, 07, 28));
23        surrealists.Add(p1.LastName, p1);
24        surrealists.Add(p2.LastName, p2);
25        surrealists.Add(p3.LastName, p3);
26        surrealists.Add(p4.LastName, p4);
27        surrealists.Add(p5.LastName, p5);
28     }
29
30     public Person LookUp(String last_name) {
31        Person p = null;
32        try {
33           if (surrealists.TryGetValue(last_name, out p)) {
34              return p;
35              }
36           else {
37                throw new SurrealistNotFoundException("That name is not in the surrealist collection!");
38           }
39        }
40        catch (ArgumentNullException ane) {
41           throw new SurrealistNotFoundException("A null string name was entered!", ane);
42        }
43     }
44  } // end SurrealistBank class definition
```

Referring to Example 15.6 — as stated in the comments at the top of the source file, this class depends on the Person class. When you compile this example, be sure to copy over the Person class from one of the examples given in Chapter 14.

The SurrealistBank class creates a Dictionary and populates it with Person objects. Its LookUp() method searches the dictionary for a matching key value via the Dictionary.TryGetValue() method. Note how the Person reference p is passed as an `out` parameter to the method call. The TryGetValue() method returns the boolean value *true* if there's a key match; *false* otherwise. If there's a match, the LookUp() method returns p.

15.7 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4       public static void Main() {
5          SurrealistBank sb = new SurrealistBank();
6          String input = "Go";
7          while (!input.Equals("Quit")) {
8             Console.Write("Please enter a name to lookup or 'Quit' to exit the program: ");
9             input = Console.ReadLine();
10               try {
11                  Console.WriteLine();
12                  Console.WriteLine(sb.LookUp(input));
13               } catch (SurrealistNotFoundException snfe) {
14                     Console.WriteLine(snfe.Message);
15                  }
16         }//end while
17      }
18   }
```

Referring to Example 15.7 — when the program executes, the input string is initialized to "Go". The `while` loop beginning on like 7 repeats until the user enters "Quit". All strings read from the console are presented to the sb.LookUp() method, which may throw a SurrealistNotFoundException. Figure 15-8 shows the results of running this program.

Manually Throwing An Exception With The throw Keyword

Referring to Example 15.6 — note again that if there's a match, the LookUp() method returns p, otherwise, it throws the SurrealistNotFoundException. It does this by using the `throw` keyword in conjunction with the creation of a new SurrealistNotFoundException object.

Translating Low-Level Exceptions Into High-Level Exceptions

Referring again to Example 15.6 — the second `catch` block beginning on line 40 handles the ArgumentNullException. This exception may be thrown by the Dictionary.TryGetValue() method if the supplied key object is null. In this example, the ArgumentNullException supplied by the .NET Framework is translated into a higher-level excep-

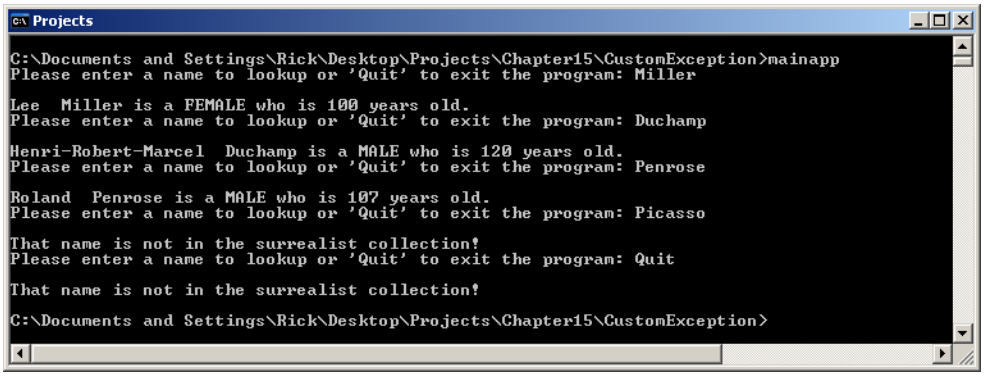

Figure 15-8: Results of Running Example 15.7

tion abstraction by throwing a SurrealistNotFoundException and using the ArgumentNullException object to set its InnerException property via the constructor call.

Quick Review

Create a *custom exception* by extending the Exception class and providing implementations for each of its three common constructors. End the name of your custom exception with the word "Exception". Use the `throw` keyword to throw an exception object.

Documenting Exceptions

You must consult the MSDN, or your locally generated documentation, to learn what exceptions, if any, a method might throw upon execution. To document an exception use the <exception> tag. Example 15.8 shows how the <exception> tag is used to document the exception thrown by the SurrealistBank.LookUp() method.

15.8 Documented SurrealistBank.LookUp() Method

```
1    /// <summary>
2    /// Searches the SurrealistBank's dictionary for the given name. Returns a populated
3    /// Person object if there's a match. Throws a SurrealistNotFoundException if no
4    /// match is found or if a null string is used as an argument.
5    /// </summary>
6    /// <param name="last_name"> A string representing a surrealist's last name.</param>
7    /// <returns>Fully populated Person object.</returns>
8    /// <exception cref="SurrealistNotFoundException"></exception>
9    public Person LookUp(String last_name) {
10      Person p = null;
11      try {
12         if (surrealists.TryGetValue(last_name, out p)) {
13            return p;
14         }
15         else {
16            throw new SurrealistNotFoundException("That name is not in the surrealist collection!");
17         }
18      } catch (ArgumentNullException ane) {
19            throw new SurrealistNotFoundException("A null string name was entered!", ane);
20         }
21   }
```

Referring to Example 15.8 — on line 8, the <exception> tag is used to document the possibility that the LookUp() method may throw the SurrealistNotFoundException. The `cref` attribute provides a link to the SurrealistNotFoundException class. The name of the class should be fully namespace qualified.

Summary

An *exception* is an error condition or abnormal event that occurs during program execution.

The .NET runtime handles exceptions by creating an exception object, populating it with information related to the error condition that occurred, and then passing it on to any fault-handling code that has been designated to respond to that particular type of exception. This process is referred to as "throwing" an exception.

Protected code is located in a `try` block. Error handlers are placed in a `catch` block.

Each executable file loaded into the .NET runtime contains an *exception information table*. All methods contained within the executable file have an entry in the exception information table. When an exception occurs, a method's corresponding handler information array is searched for any associated error handlers. If none are found, the exception is propagated up the method call stack to search the calling method's handler information array. If no handler methods are found, the .NET runtime dumps the stack trace to the console and the program terminates.

Two primary types of exceptions derive from the Exception class: *SystemException* and *ApplicationException*. *Runtime exceptions* occur for two primary reasons: 1) failed runtime checks, and 2) severe error conditions that occur within the runtime execution environment. An *application exception* is any exception thrown by a running program that's not a runtime exception. Application exceptions derive from either the ApplicationException class or directly from the Exception class itself. Consult the Exceptions section of a method's MSDN documentation page to learn what exceptions a method might throw.

To catch and handle exceptions, you need to place exception-throwing code in a `try` block and catch the resulting exception in one or more `catch` blocks where error-handling code is placed to properly deal and recover from the exception. Optionally, you can add a `finally` block whose code is always executed regardless of if an exception is thrown or not.

All code within a `try` block that follows an exception-throwing statement is skipped. Place critical resource releasing code in a `finally` block.

If your code can throw several types of exceptions, you can add multiple `catch` blocks, one for each exception type. The rule to follow when using multiple `catch` blocks is to catch the most specific exception(s) first, and catch the most general exception(s) last.

Create a *custom exception* by extending the Exception class and providing implementations for each of its three common constructors. End the name of your custom exception with the word "Exception". Use the `throw` keyword to throw an exception object.

Skill-Building Exercises

1. **API Drill:** Access the MSDN online documentation and explore the exception classes defined within the System namespace. Note their purpose and under what circumstances they might be thrown.

2. **API Drill:** Access the MSDN online documentation and explore the exception classes defined within the System.Collection and System.Collection.Generic namespaces. Note their purpose and under what circumstances they might be thrown.

3. **API Drill:** Access the MSDN online documentation and explore the exception classes defined within the System.IO namespace. Note their purpose and under what circumstances they might be thrown.

4. **API Drill:** Access the MSDN online documentation and explore the exception classes defined within the System.Net namespace. Note their purpose and under what circumstances they might be thrown.

Suggested Projects

1. None

Self-Test Questions

1. What are the two meanings of the term *exception*?

2. How does the .NET runtime deal with exceptions?

3. How does the .NET runtime keep track of protected code and associated exception handlers?

4. What are the two primary types of exceptions defined my the .NET Framework?

5. What's the difference between a RuntimeException and an ApplicationException?

6. In what cases might a RuntimeException be thrown?

7. (T/F) Microsoft recommends extending the ApplicationException class to create a custom exception?

8. What happens to the code in a `try` block that follows a statement that just threw an exception?

9. Where should you place critical resource-freeing code or code that must be executed regardless of whether or not an exception is thrown?

10. What document tag do you use to indicate what exceptions a method can throw?

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Notes