CHAPTER 5

Exceptions and Object Lifetime

In the previous chapter, you dug into the details of how C# supports encapsulation services, reuse of code (inheritance), and polymorphic activity among related classes. The point of this chapter is twofold. First, you will come to understand how to handle runtime anomalies in your code base through the use of structured exception handling. Not only will you learn about the C# keywords that allow you to handle such problems (“try,” “catch,” “throw,” “finally”), but you will also come to understand the distinction between application-level and system-level exceptions. This discussion will also provide a lead-in to the topic of building custom exception types, as well as how to leverage the built-in exception handling functionality of Visual Studio .NET.

The second half of the chapter will examine the process of object lifetime management as handled by the CLR. As you will see, the .NET runtime destroys objects in a rather nondeterministic nature. Thus, you typically do not know when a given object will be deallocated from the managed heap, only that it will (eventually) come to pass. On this note, you will come to understand how the System.Object.Finalize() method and IDisposable interface can be used to interact with an object’s lifetime management. Finally, we wrap up with an examination of the System.GC type, and illustrate a number of ways you are able to “get involved” with the garbage collection process.

Ode to Errors, Bugs, and Exceptions

Despite what our (often inflated) egos may tell us, no programmer is perfect. Writing software is a complex undertaking, and given this complexity, it is quite common for even the best software to ship with various “problems.” Sometimes the problem is caused by “bad code” (such as overflowing the bounds of an array). Other times, a problem is caused by bogus user input that has not been accounted for in the application’s code base (e.g., a phone number field assigned “Chucky”). Now, regardless of the cause of said problem, the end result is that your application does not work as expected. To help frame the upcoming discussion of structured exception handling, allow me to provide definitions for three common anomaly-centric terms:

- **Bugs**: This is, simply put, an error on the part of the programmer. For example, assume you are programming with unmanaged C++. If you make calls on a NULL pointer, overflow the bounds of an array, or fail to delete allocated memory (resulting in a memory leak), you have a bug.
• **Errors:** Unlike bugs, errors are typically caused by the end user of the application, rather than by those who created said application. For example, an end user who enters a malformed string into a text box that requires a Social Security number could very well generate an error if you fail to trap this faulty input in your code base.

• **Exceptions:** Exceptions are typically regarded as runtime anomalies that are difficult, if not impossible, to prevent. Possible exceptions include attempting to connect to a database that no longer exists, opening a corrupted file, or contacting a machine that is currently offline. In each of these cases, the programmer (and end user) has little control over these “exceptional” circumstances.

Given the previous definitions, it should be clear that .NET structured exception handling is a technique best suited to deal with runtime exceptions. However, as for the bugs and errors that have escaped your view, do be aware that the CLR will generate a corresponding exception that identifies the problem at hand. As you will see, the .NET base class libraries define exceptions such as OutOfMemoryException, IndexOutOfRangeException, FileNotFoundException, ArgumentOutOfRangeException, and so forth.

Before we get too far ahead of ourselves, let’s formalize the role of structured exception handling and check out how it differs from traditional error handling techniques.

### The Role of .NET Exception Handling

Error handling among Windows developers has grown into a confused mishmash of techniques over the years. Many programmers roll their own error handling logic within the context of a given application. For example, a development team may define a set of numerical constants that represent known error conditions, and make use of them as method return values. For example, ponder the following C code:

```c
/* A very C-style error trapping mechanism. */
#define E_FILENOTFOUND 1000

int SomeFunction()
{
    // Assume something happens in this f(x)
    // which causes the following return value.
    return E_FILENOTFOUND;
}

void Main()
{
    int retVal = SomeFunction();
    if(retVal == E_FILENOTFOUND)
        { printf("Can not find file...");}
}
```

This approach is less than ideal, given the fact that the constant E_FILENOTFOUND is little more than a raw numerical value, and is far from being a self-describing agent.