Programming in C#

E04 C# Exception and debug

Yu Guan | Microsoft MVP
2018

AGENDA

- Collections
- Exception handling
- Validation
- Debug an application
- Implement diagnostics

Collections

- Use the collections that are part of the .NET Framework
- Choose which collection to use
- Implement your own collections

Using arrays

- Arrays are useful when you are working with a fixed number of objects that all have the same type.
- The biggest problem with arrays is that they are of fixed size. When working with groups of objects, you often want to add or remove items from the collection. This is why the .NET Framework has some other collection types.

```
int[] arrayOfInt = new int[10];
```

Using List<T>

- One collection type you will probably use most often is the generic List<T> collection. The List type offers methods for adding and removing items, accessing items by index, and searching and sorting the list.
- The List type makes sure that there is always enough room to store additional items. If necessary, the internal implementation of the List class will increase the size of the array it uses to store its items. List<T> can store reference types and it can have a value of null for an item. It can also store duplicate items.

```
List<string> listOfStrings =
    new List<string> { “A”, “B”, “C”, “D”, “E” , “E” };
```

Using Dictionary

- A Dictionary<TKey,TValue> can be used in scenarios in which you want to store items and retrieve them by key, so it doesn't allow duplicate keys. It takes two type parameters: one for the type of the key, and the other for the type of the value.
- The Dictionary class is implemented as a hash table, which makes retrieving a value very fast, close to O(1). The hash value of a key shouldn't change during time and it can't be null. The value can be null (if it's a reference type).

```
var dict = new Dictionary<int, Person>();
```

Using sets

- In some languages, such as Java, there is a special set type. In C#, a set is a reserved keyword, but you can use the HashSet<T> if you need one. A set is a collection that contains no duplicate elements and has no particular order.

```
HashSet<int> oddSet = new HashSet<int>();
```

Using queues

- A queue is a special type of collection you can use to temporarily store some data. It is a so called first-in, first-out (FIFO) type of collection, just like a checkout line.
- The Queue class has three important methods:
  - Enqueue adds an element to the end of the Queue, equivalent to the back of the line.
  - Dequeue removes the oldest element from the Queue, equivalent to the front of the line.
  - Peek returns the oldest element, but doesn't immediately remove it from the Queue.

```
Queue<string> myQueue = new Queue<string>();
```

Using stacks

- Stack is a last-in, first-out (LIFO) collection. Think of the undo system of an application. The last item added to the undo stack is the first one to be used when a user executes an undo action. Just as with a Queue, items are removed when reading them.
- A Stack has the following three important methods:
  - Push Add a new item to the Stack
  - Pop Get the newest item from the Stack
  - Peek Get the newest item without removing it

```
Stack<string> myStack = new Stack<string>();
```

Choosing a collection

- When choosing a collection type, you have to think about the scenarios you want to support. The biggest differences between the collections are the ways that you access elements.
- List and Dictionary types offer random access to all elements. A Dictionary offers faster read features, but it can't store duplicate elements.
- A Queue and a Stack are used when you want to retrieve items in a specific order. The item is removed when you have retrieved it.
- Set-based collections have special features for comparing collections. They don't offer random access to individual elements.
- Although List can be used in most situations, it pays to see whether there is a more specialized collection that can make your life easier.

Creating a custom collection

- IList<T>
- ICollection<T>
- IDictionary<TKey,TValue>
- ICollection<TKey,TValue>
- ISet<T>
- directly inherit from an existing collection

Creating a custom collection

```
public class PeopleCollection : List<Person>
{
    public void RemoveByAge(int age)
    {
        for (int index = this.Count - 1; index >= 0; index--)
        {
            if (this[index].Age == age)
            {
                this.RemoveAt(index);
            }
        }
    }
    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        foreach (Person p in this)
        {
            sb.AppendFormat("{0} {1} is {2}", p.FirstName, p.LastName, p.Age);
        }
        return sb.ToString();
    }
}
```

Implement exception handling

- Handle exceptions
- Throw exceptions
- Create custom exceptions

Handling exceptions

- When an error occurs somewhere in an application, an exception is raised. Exceptions have a couple of advantages compared with error codes. An exception is an object in itself that contains data about the error that happened. It not only has a user-friendly message but it also contains the location in which the error happened and it can even store extra data, such as an address to a page that offers some help.

```
try
{
    int i = int.Parse(s);
    break;
}
catch (FormatException)
{
    Console.WriteLine("{0} is not a valid number. Please try again", s);
}
```

Using a finally block

```
try
{
    int i = int.Parse(s);
}
catch (ArgumentNullException)
{
    Console.WriteLine(“You need to enter a value”);
}
catch (FormatException)
{
    Console.WriteLine(“{0} is not a valid number. Please try again”, s);
}
finally
{
    Console.WriteLine(“Program complete.”);
}
```

Throwing exceptions

- When you want to throw an error, you first need to create a new instance of an exception. You can then use the special throw keyword to throw the exception. After this, the runtime will start looking for catch and finally blocks.
- Make sure that you don't swallow any exception details when rethrowing an exception. Throw a new exception that points to the original one when you want to add extra information; otherwise, use the throw keyword without an identifier to preserve the original exception details.

```
public static string OpenAndParse(string fileName)
{
    if (string.IsNullOrWhiteSpace(fileName))
        throw new ArgumentNullException("fileName", "Filename is required");
    return File.ReadAllText(fileName);
}
```

Creating custom exceptions

- Once throwing an exception becomes necessary, it's best to use the exceptions defined in the .NET Framework. But there are situations in which you want to use a custom exception. This is especially useful when developers working with your code are aware of those exceptions and can handle them in a more specific way than the framework exceptions.
- A custom exception should inherit from System.Exception. You need to provide at least a parameterless constructor. It's also a best practice to add a few other constructors.

```
[Serializable]
public class OrderProcessingException : Exception, Iserializable
{
}
```

Validate application input

- Explain why validating application input is important
- Use Parse, TryParse, and Convert
- Use regular expressions for input validation

Why validating application input is important

- When your application is in production, it has to deal with various types of input. Some of this input comes from other systems that it integrates with, and most input is generated by users.
- Those users fall into two categories:
  - Innocent users
  - Malicious users

Using Parse, TryParse, and Convert

- The Parse and TryParse methods can be used when you have a string that you want to convert to a specific data type.
- The .NET Framework also offers the Convert class to convert between base types. The supported base types are Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, Uint32, Uint64, Single, Double, Decimal, DateTime, and String.
- The difference between Parse/TryParse and Convert is that Convert enables null values. It doesn't throw an ArgumentNullException; instead, it returns the default value for the supplied type.

Using regular expressions

- A regular expression is a specific pattern used to parse and find matches in strings.
- A regular expression is sometimes called regex or regexp.
- https://regexr.com/

```
static bool ValidateZipCodeRegEx(string zipCode)
{
    Match match = Regex.Match(zipCode, @"^[1-9][0-9]{3}\s?[a-zA-Z]{2}$",
        RegexOptions.IgnoreCase);
    return match.Success;
}
```

Debug an application

- Choose an appropriate build type
- Create and manage compiler directives
- Manage program database files and symbols

Build configurations

- If you create a new project in Visual Studio, it creates two default build configurations for you
  - Release mode
  - Debug mode
- If you compile your project, the settings from these configurations are used to configure what the compiler does.
- In release mode, the compiled code is fully optimized, and no extra information for debugging purposes is created.
- In debug mode, there is no optimization applied, and additional information is outputted.

Creating and managing compiler directives

- Some programming languages have the concept of a preprocessor, which is a program that goes through your code and applies some changes to your code before handing it off to the compiler.
- C# does not have a specialized preprocessor, but it does support preprocessor compiler directives, which are special instructions to the compiler to help in the compilation process.

```
public void DebugDirective()
{
    #if DEBUG
        Console.WriteLine(“Debug mode”);
    #else
        Console.WriteLine(“Not debug”);
    #endif
}
```

Managing program database files and symbols

- When compiling your programs, you have the option of creating an extra file with the extension .pdb. This file is called a program database (PDB) file, which is an extra data source that annotates your application's code with additional information that can be useful during debugging.
- You can construct the compiler to create a PDB file by specifying the /debug:full or /debug:pdbonly switches. When you specify the full flag, a PDB file is created, and the specific assembly has debug information. With the pdbonly flag, the generated assembly is not modified, and only the PDB file is generated. The latter option is recommended when you are doing a release build.
- A .NET PDB file contains two pieces of information:
  - Source file names and their lines
  - Local variable names
- This data is not contained in the .NET assemblies, but you can imagine how it helps with debugging.

Implement diagnostics

- Implement logging and tracing.
- Profile your applications.
- Create and monitor performance counters.

Logging and tracing

- When your application is running on a production server, it's sometimes impossible to attach a debugger because of security restrictions or the nature of the application. If the application runs on multiple servers in a distributed environment, such as Windows Azure, a regular debugger won't always help you find the error.

```
Debug.WriteLine("Starting application");
Debug.Indent();
int i = 1 + 2;
Debug.Assert(i == 3);
Debug.WriteLineIf(i > 0, "i is greater than 0");
```

Profiling your application

- When looking for a performance problem, the only real way to find it is measure, not guess. Maybe you have a feeling where the problem is but just making some random changes and then verifying that your applications performance has improved is really hard.
- A simple way of measuring the execution time of some code is by using the Stopwatch class that can be found in the System.Diagnostics namespace.

Creating and monitoring performance counters

```
namespace PerformanceCounters
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(“Press escape key to stop”);
            using (PerformanceCounter pc = new PerformanceCounter(“Memory”, “Available Bytes”))
            {
                string text = “Available memory: “;
                Console.Write(text);
                do
                {
                    while (!Console.KeyAvailable)
                    {
                        Console.Write(pc.RawValue);
                        Console.SetCursorPosition(text.Length, Console.CursorTop);
                    }
                } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
            }
        }
    }
}
```

Recap

- Exam Ref 70-483
  - CHAPTER 1 Manage program flow
    - Objective 1.1: Implement multithreading and asynchronous processing
    - Objective 1.2: Manage multithreading
    - Objective 1.5: Implement exception handling
  - CHAPTER 3 Debug applications and implement security
    - Objective 3.1: Validate application input
    - Objective 3.4: Debug an application
    - Objective 3.5. Implement diagnostics in an application
- CSharp Language Specification
  - 16

THANK YOU!

Q&A