Advanced Programming in C#

Rex Jaeschke

Advanced Programming in C#

Edition: 2.0

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

This text covers a number of more advanced C# topics. The material is not hardware or operating system-specific.

Reader Assumptions

To fully understand and exploit the material, you should be conversant with the following concepts and the syntax required to express them in C#:

- Basic Language Elements
- Looping and Testing
- Methods
- References
- Strings and Arrays
- Classes
- Inheritance
- Interfaces
- Exception Handling
- Input and Output
- Namespaces

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming-language training courses for more than 15 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and well focused. Specifically, I introduce the basic elements and constructs of the language using procedural programming examples. Once those fundamentals have been mastered, I move on to object-oriented concepts and syntax. Then follow the more advanced language and library topics. Many books on object-oriented languages use objects, inheritance, exception handling, GUI, and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than is any other; I simply know that my approach works well, and has formed the basis of my successful seminar business for more than a decade.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory. By convention, the names of C# source files end in "`.cs`".

Each chapter contains exercises, some of which have the character ***** following their number. For each exercise so marked, a solution is provided in a directory tree named Labs, in which each chapter has its own subdirectory.[1] Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Exercises having solutions contain a statement of the form "(See lab directory *xx*.)", which indicates the corresponding solution or test file in the Labs subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

The Status of C#

Microsoft announced the C# language and .NET platform in July 2000. At that time, Microsoft stated that both the language and some subset of the library and runtime environment would be submitted for standardization, to Ecma International, a standards organization. In September 2000, Ecma committee TC39, which was responsible for standardization of the scripting language ECMAScript (known commercially as JavaScript and JScript), agreed to take on this new work. As a result, TC39 was split into three Task Groups: TG1 (ECMAScript), TG2 (C#), and TG3 (Common Library Infrastructure [CLI]). This author served as project editor of the standards produced by TG2 and TG3.

The submission to Ecma was sponsored by Hewlett-Packard, Intel, and Microsoft. A number of other companies also agreed to participate in the standards work, which began in November 2000. That work was completed in September 2001, and both standards were adopted by Ecma in December 2001. The standard for C# is ECMA-334, and that for CLI is ECMA-335. (These standards can be downloaded free of charge from www.ecma-international.org.)

In January 2002, a 6-month review and ballot period began within ISO/IEC JTC 1 to determine if these specifications should be adopted as ISO/IEC standards as well. After a ballot-resolution meeting in September 2002, these specifications were approved unanimously, and they were forwarded to ISO for publication. The designation of the C# standard is ISO/IEC 23270, while that for the CLI standard is ISO/IEC 23271. (See www.iso.org.)

A revision of the Ecma versions of these standards started in January 2003, and was completed in March 2005. Ecma adopted them in June 2005, and ISO/IEC JTC 1 adopted them in April 2006. This revision included a number of important additions: generic types and methods, static classes, anonymous methods, partial declarations, iterators, nullable types, and pragma directives.

Throughout this text, the original definition of C# is referred to as V1, while the revised version is referred to as V2. (Microsoft's current version, V3, is not covered by this book or the C# Standard.)

---

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

In 2008, Ecma transferred the Task Groups TG2 (C#) and TG3 (CLI) from Technical Committee TC39 to TC49.

Acknowledgments

Many thanks go to those people who reviewed all or part of this book. In particular, Pat Bria, and students in my introductory C# seminars provided useful feedback and located numerous typographical errors. Thanks also to the folks at Microsoft for their assistance, especially to Peter Golde, Peter Hallam, and Scott Wiltamuth for help on language issues, and Brad Abrams for help with questions regarding the class libraries.

Rex Jaeschke, September 2009

1. Threads

C# supports the ability to create multiple threads of execution within a single program. In this chapter, we'll see how threads are created and synchronized.[1] We'll also see how shared variables can be guarded against compromise during concurrent operations.

1.1 Introduction

A *thread* is an individual stream of execution as seen by the processor, and each thread has its own register and stack context. The run-time environment executes only one thread at a time. The execution of a thread is interrupted when it needs resources that are not available, it is waiting for an operation such as an I/O to complete, or if it uses up its processor time slice. When the processor changes from executing one thread to another, this is called *context switching*. By executing another thread when one thread becomes blocked, the system allows processor idle time to be reduced. This is called *multitasking*.

When a program is executed, the system is told where on disk to get instructions and static data. A set of virtual memory locations, collectively called an *address space* is allocated to that program, as are various system resources. This runtime context is called a *process*. However, before a process can do any work, it must have at least one thread. When each process is created, it is automatically given one thread, called the *primary thread*. However, this thread has no more capability than other threads created for that process; it just happened to the first thread created for that process. The number of threads in a process can vary at runtime, under program control. Any thread can create other threads; however, a creating thread does not in any sense own the threads it creates; all threads in a process belong to the process as a whole.

The work done by a process can be broken into subtasks with each being executed by a different thread. This is called *multithreading*. Each thread in a process shares the same address space and process resources. When the last remaining thread in a process terminates, the parent process terminates.

Why might we want to have more than one thread in a process? If a process has only one thread, it executes serially. When the thread is blocked, the system is idle if no other process has an active thread waiting. This may be unavoidable if the subtasks of the process must be performed serially; however, this is not the case with many processes. Consider a process that has multiple options. A user selects some option, which results in lots of computations using data in memory or a file and the generation of a report. By spawning off a new thread to perform this work, a process can continue accepting new requests for work without waiting for the previous option to complete. And by specifying thread priorities, a process can allow less-critical threads to run only when more-critical threads are blocked.

Once a thread has been dispatched, another thread can be used to service keyboard or mouse input. For example, the user might decide that a previous request is not the way to go after all, and wishes to abort the first thread. This can be done by selecting the appropriate option on a pull-down menu and having one thread stop the other.

[1] It is important to note that C# does not support synchronization of threads in different programs.

Another example involves a print spooler. Its job is to keep a printer busy as much as possible and to service print requests from users. The users would be very unhappy if the spooler waits until a job had completed printing before it started accepting new requests. Of course, it could periodically stop printing to see if any new requests were pending (this is called *polling*), but that wastes time if there are no requests. And if the time interval between polls is too long, there is a delay in servicing requests. If it is too short, the thread spends too much time polling. Why not have the spooler have two threads: one to send work to the printer, the other to deal with requests from users. Each runs independent of the other, and when a thread runs out of work, it either terminates itself or goes into an efficient state of hibernation.

When dealing with concurrently executing threads, we must understand two important aspects: atomicity and reentrancy.

An *atomic* variable or object is one that can be accessed as a whole even in the presence of asynchronous operations that access the same variable or object. For example, if one thread is updating an atomic variable or object while another thread reads its contents, the logical integrity of those contents cannot be compromised—the read will get either the old or the new value, never part of each. Normally, the only things that can be accessed atomically are those having types supported atomically in hardware, such as bytes and words. All the primitive types in C#, except `long`, `decimal`, and `double`, are guaranteed to be atomic. (These might also be atomic for a given implementation, however, that's not guaranteed.) Clearly, a Point object is not atomic; it has two parts, an x- and a y-coordinate, and a writer of a Point's value could be interrupted by a reader to that Point, resulting in the reader getting the new x and old y, or vice-versa. Similarly, arrays cannot be accessed atomically. Since most objects cannot be accessed atomically, we must use some form of synchronization to ensure that only one thread at a time can operate on certain objects. For this reason, C# assigns each object, array, and class a *synchronization lock*.

A *reentrant method* is one, which can safely be executed in parallel by multiple threads of execution. When a thread begins executing a method, all data allocated in that method comes either from the stack or from the heap. In any event, it's unique to that invocation. If another thread begins executing that same method while the first thread is still working there, each thread's data will be kept separate. However, if that method accesses variables or files that are shared between threads, it must use some form of synchronization.

1.2 Creating Threads

In the following example (see directory Th01), the primary thread creates two other threads, and the three threads run in parallel without synchronization. No data is shared between the threads and the process terminates when the last thread terminates:

```
using System;
using System.Threading;

class Th01
{
        private int loopStart;
        private int loopEnd;
        private int dispFrequency;
```



```
        public Th01(int startValue, int endValue, int frequency)
        {
                loopStart = startValue;
                loopEnd = endValue;
                dispFrequency = frequency;
        }

/*1*/   public void ThreadEntryPoint()
        {
/*2*/           string threadName = Thread.CurrentThread.Name;

                for (int i = loopStart; i <= loopEnd; ++i)
                {
                        if (i % dispFrequency == 0)
                        {
                                Console.WriteLine("{0}: i = {1,10}",
                                        threadName, i);
                        }
                }
                Console.WriteLine("{0} thread terminating", threadName);
        }

        public static void Main()
        {
/*3a*/          Th01 o1 = new Th01(0, 1000000, 200000);
/*3b*/          Thread t1 = new Thread(new ThreadStart(o1.ThreadEntryPoint));
/*3c*/          t1.Name = "t1";

/*4a*/          Th01 o2 = new Th01(-1000000, 0, 200000);
/*4b*/          Thread t2 = new Thread(new ThreadStart(o2.ThreadEntryPoint));
/*4c*/          t2.Name = "t2";

/*5*/           t1.Start();
/*6*/           t2.Start();
                Console.WriteLine("Primary thread terminating");
        }
}
```

Let's begin by looking at the first executable statement in the program, that in case 3a. Here we create an object having the user-defined type `Th01`. That class has a constructor, an instance method, and three fields. We call the constructor passing it a start and end count, and an increment amount, which it stores for later use in controlling a loop.

In case 3b, we create an object of the library type `Thread`, which is from the namespace `System.Threading`. A new thread is created using such an object; however, before a thread can do useful work, it must know where to start execution. We indicate this by passing to `Thread`'s constructor a delegate of type `ThreadStart`, which

2. Object Serialization

Most useful applications depend on information of a more permanent nature than that generated during a single execution; for example, applications that access an inventory, typically query (and possibly update) one of more related data files. The lives of such "master files" transcend that of the execution of any of the applications that use them. Other applications involve the communication of messages between separate programs, often referred to as *client* and *server*. While the life of a message is often much shorter than that of a database record, both involve the use of some data format external to the applications that manipulate them.

In this chapter, we'll see how C# objects and variables of simple types can be converted into some external form suitable for use in file storage or for transmission during inter-application communication. The process of converting to some external form is known as *serialization* while that of converting back again is known as *deserialization*.[1]

2.1 Introduction

Consider the following example (see directory Sr01) which writes a number of values of a variety of object and simple types to a disk file, closes that file, and then reads those values back into memory again:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

class Sr01
{
        public static void Main()
        {
                int[] intArray = {10, 20, 30};
                float[,] floatArray = {
                        {1.2F, 2.4F},
                        {3.5F, 6.8F},
                        {8.4F, 9.7F}
                };
                DateTime dt = DateTime.Now;
                Console.WriteLine("dt      >{0}<", dt);

/*1*/           BinaryFormatter formatter = new BinaryFormatter();
```

[1] Support for serialization and deserialization is *not* included in the CLI standard. However, it is included in Microsoft's Visual C# implementation.

In case 1, we define a variable of type `BinaryFormatter`. Objects of this type allow the serialization and deserialization of any object or an entire graph[1] of connected objects in some binary format. (We'll see in §2.6 how alternate formats can be used.)

```
        // Serialize data to a file.

/*2*/           Stream file = File.Open("Sr01.ser", FileMode.Create);

/*3a*/          formatter.Serialize(file, "Hello");
/*3b*/          formatter.Serialize(file, intArray);
/*3c*/          formatter.Serialize(file, floatArray);
/*3d*/          formatter.Serialize(file, true);
/*3e*/          formatter.Serialize(file, dt);
/*3f*/          formatter.Serialize(file, 1000);
/*3g*/          formatter.Serialize(file, 'X');
/*3h*/          formatter.Serialize(file, 1.23456F);

/*4*/           file.Close();
```

In case 2, we create a new file having the name shown. The suffix ".ser" has no special meaning; it is simply a local convention that signifies a serialized data file. Cases 3a through 3h each result in an object being serialized to that file. In the case of the string, each character is written. In the case of the arrays, all elements are written. In the case of the DateTime object, all data contained within an object of that type and any associated dependencies is written. In the case of the simply type values, they are first boxed and the corresponding `object`s are written. So `Serialize` need only be defined to accept an argument of type `object`.

```
        // Deserialize data from a file.

/*5*/           file = File.Open("Sr01.ser", FileMode.Open);

/*6a*/          string s = formatter.Deserialize(file) as string;
                Console.WriteLine("String  >{0}<", s);
```

We retrieve the serialized data by calling the method `Deserialize`, as shown in case 6a. Since that method returns a value of type `object`, we need to convert that to the appropriate type, which we do via the `as` operator. This operator is used to explicitly convert a value to a given reference type using a reference conversion or a boxing conversion. Unlike a cast expression, the `as` operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is `null`.

[1] The term *graph* as used here refers to a kind of nonlinear data structure. For example, an Employee type might contain a name field, which, in turn, is made up of a first name, an initial, and a last name. The type might also contain several date objects, representing date of birth, date of hire, and date of last review. The graph designated by a reference to an instance to such a type involves that instance's entire dependent data tree starting at the root reference.


```
/*6b*/          int[] newIntArray = formatter.Deserialize(file) as int[];
                Console.WriteLine("int[]:");
                for (int i = 0; i < newIntArray.Length; ++i)
                {
                        Console.Write("  {0}", newIntArray[i]);
                }
                Console.WriteLine();

/*6c*/          float[,] newFloatArray = formatter.Deserialize(file) as float[,];
                Console.WriteLine("newFloatArray:");
                for (int i = 0; i < 3; ++i)
                {
                        for (int j = 0; j < 2; ++j)
                        {
                                Console.Write("  {0}", newFloatArray[i,j]);
                        }
                        Console.WriteLine();
                }

/*6d*/          bool b = (bool)formatter.Deserialize(file);
                Console.WriteLine("bool    >{0}<", b);

/*6e*/          DateTime newDT = (DateTime)formatter.Deserialize(file);
                Console.WriteLine("newDT   >{0}<", newDT);

/*6f*/          int v = (int)formatter.Deserialize(file);
                Console.WriteLine("int     >{0}<", v);

/*6g*/          char c = (char)formatter.Deserialize(file);
                Console.WriteLine("char    >{0}<", c);

/*6h*/          float f = (float)formatter.Deserialize(file);
                Console.WriteLine("float   >{0}<", f);

/*7*/           file.Close();
        }
}
```

In cases 6d, 6f, 6g, and 6h, we use an explicit cast for the conversion since the `as` operator can only be used to convert to a reference type. Note, however, that we also have used an explicit cast in case 6e, to convert to the reference type `DateTime`. In this case, it's simply a matter of style as to which approach is used. Both approaches work and both provide a way to handle failure.

4. Cloning Objects

Unlike most other languages, C# provides no way to have an expression of some object type; the best we can do is to have an expression whose value is really a reference to that type. As a result, C# provides no linguistic way to copy an object. Assuming we can justify it, how then can we make a complete copy of an object?

4.1 Copying by Constructor

Consider the following code fragment (see Cn01.cs):

```
class Point
{
        private int xor;
        private int yor;

        public int X { get { return xor; }  set { xor = value; }}
        public int Y { get { return yor; }  set { yor = value; }}

        public Point(Point p)
        {
                X = p.X;
                Y = p.Y;
        }
```

The constructor creates a new Point using the contents of an existing one. (In C++, this kind of constructor is so special it has the name *copy constructor*; however, this term is not used in the context of C#.)

```
class Cn01
{
        public static void Main()
        {
/*1*/           Point p1 = new Point(3, 5);
/*2*/           Console.WriteLine("p1: {0}", p1);

/*3*/           Point p2 = new Point(p1);

/*4*/           p1.Move(9, 11);

/*5*/           Console.WriteLine("p1: {0}", p1);
/*6*/           Console.WriteLine("p2: {0}", p2);
        }
}
```

Point p2 is a copy of p1, so when the coordinate values of p1 are changed in case 4, we see from the following output that those of p2 are not:

```
p1: (3,5)
p1: (9,11)
p2: (3,5)
```

While this approach works fine, we won't find standard library classes having this form of constructor, however.

4.2 Class Cloning

Those few standard library classes that do support some sort of copy mechanism achieve it by what is referred to as cloning. Consider the following example (see directory Cn02) that uses the library class `ArrayList`, a type that supports vectors:

```
using System;
using System.Collections;

class Cn02
{
        public static void Main()
        {
                ArrayList al1 = new ArrayList();

/*1*/           al1.Add("Red");
                al1.Add("Blue");
                al1.Add("Green");
                al1.Add("Yellow");
/*2*/           PrintEntries("al1", al1);

/*3*/           ArrayList al2 = (ArrayList)al1.Clone();
/*4*/           PrintEntries("al2", al2);

/*5*/           al1.Remove("Blue");
                al1.Add("Black");
                al1.RemoveAt(0);
                al1.Insert(0, "Brown");

/*6*/           PrintEntries("al1", al1);
/*7*/           PrintEntries("al2", al2);
        }
```

```
        private static void PrintEntries(string s, ArrayList aList)
        {
                Console.Write("{0}: ", s);
                foreach(object o in aList)
                {
                        Console.Write("\t{0}", o);
                }
                Console.WriteLine();
        }
}
```

The output produced is:

```
al1:    Red     Blue    Green   Yellow
al2:    Red     Blue    Green   Yellow
al1:    Brown   Green   Yellow  Black
al2:    Red     Blue    Green   Yellow
```

ArrayList `al1` consists of a set of 4 strings specifying different colors. We then make a complete copy of that object in case 3 by calling the method `ArrayList.Clone`, so the output produced by cases 2 and 4 is the same. Then we modify `al1` by removing the second element, by adding a new element to the end, and by changing the first element's value. When we compare the output produced by cases 6 and 7, we see that the changes applied to `al1` have no affect on `al2`. Specifically, the internal reference inside of `al2` points to its own private copy of elements, not to the same set as `al1`. This is referred to as a *deep copy*, whereas simply making both ArrayLists' internal reference point to the same set of values (as would be the case with the assignment `al2 = al1`) is called a *shallow copy*.

If we wish to make copies of our own objects, we would do well to model the copy process on the library cloning machinery.

4.3 The Method `Clone`

The key to cloning is to implement the standard interface `ICloneable`, which, in turn, requires us to define a method called `Clone` that takes no arguments and has a return type of `object`. For example, consider the following program fragment (see directory Cn03):

```
using System;

class Point : ICloneable
{
        private int xor;
        private int yor;

        public int X { get { return xor; }  set { xor = value; }}
        public int Y { get { return yor; }  set { yor = value; }}
```